ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Magistrale in Scienze di Internet

WORKFLOW PATTERNS FOR SERVICE ORIENTED COMPUTING IN JOLIE

Tesi di Laurea Magistrale in Intelligenza Artificiale

Relatore:
Chiar.mo Prof.
MAURIZIO GABBRIELLI

Presentata da:
SAVERIO GIALLORENZO

Sessione 1
Anno Accademico 2011/2012

a Mamma e Papà,

miei punti di riferimento,
grazie per aver sempre creduto in me,
anche se, non sempre, avevo ragione.

a Danny,

la migliore compagna che potessi desiderare,
grazie per avermi accettato nella tua vita
e per volerla condividere con me.

Sommario

Il presente lavoro di tesi ha come punto focale la descrizione, la verifica e la dimostrazione della realizzabilità dei *Workflow Patterns* di *Gestione del Flusso* (Control-Flow) e *Risorse* (Resource) definiti da parte della *Workflow Pattern Initiative* (WPI) in JOLIE, un innovativo linguaggio di programmazione orientato ai servizi nato nell'ambito del *Service Oriented Computing*.

Il Service Oriented Computing (SOC) è un nuovo modo di pensare la programmazione di applicazioni distribuite, i cui concetti fondamentali sono i servizi e la composizione. L'approccio SOC definisce la possibilità di costruire un'applicazione in funzione dei *servizi* che ne realizzano il comportamento tramite una loro *composizione*, definita secondo un particolare flusso di lavoro.

Allo scopo di fornire la necessaria conoscenza per capire la teoria, le meccaniche e i costrutti di JOLIE utilizzati per la realizzazione dei pattern, il seguente lavoro di tesi è stato diviso in quattro parti, corrispondenti ad altrettanti capitoli.

Nel **primo capitolo** viene riportata una descrizione generale del SOC e della *Business Process Automation* (BPA), che costituisce l'ambiente in cui il SOC è inserito. Per questo viene fatta una disamina della storia informatica sui sistemi distribuiti, fino ad arrivare ai sistemi odierni, presentando in seguito il contesto del BPA e delle innovazioni derivanti dalle sue macro-componenti, di cui il SOC fa parte.

Continuando la descrizione dell'approccio Service Oriented, ne vengono presentati i requisiti (pre-condizioni) e si cerca di dare una definizione precisa del termine "servizio", fino all'enunciazione dei principi SOC declinati nell'ottica delle *Service Oriented Architectures*, presentando in ultimo i metodi di composizione dei servizi, tramite *orchestrazione* e *coreografia*.

L'ultima sezione del capitolo prende in considerazione il SOC in un'ottica prettamente industriale e ne evidenzia i punti strategici.

Il **secondo capitolo** è incentrato sulla descrizione di JOLIE, gli aspetti fondamentali dell'approccio orientato ai servizi, che ne caratterizzano profondamente la definizione concettuale (SOCK), e la teoria della composizione dei servizi.

Il capitolo non si pone come una descrizione esaustiva di tutte le funzionalità del linguaggio, ma considera soprattutto i concetti teorici, le strutture di dati, gli operatori e i costrutti di JOLIE utilizzati per la dimostrazione della realizzabilità dei Workflow Pattern del capitolo successivo.

Il **terzo capitolo**, più lungo e centrale rispetto agli altri, riguarda la realizzazione dei workflow pattern in JOLIE. All'inizio del capitolo viene fornita una descrizione delle caratteristiche del WPI e dei Workflow Pattern in generale. In seguito, nelle due macro-sezioni relative ai *Control-Flow* e *Resource pattern* vengono esposte alcune nozioni riguardanti le metodologie di definizione dei pattern (e.g. la teoria sulla definizione delle *Colored Petri Nets*) e le convezioni adottate dal WPI, per passare in seguito al vero e proprio lavoro (sperimentale) di tesi riguardo la descrizione dei pattern, l'analisi sulla loro realizzabilità in JOLIE, insieme ad un codice di esempio che esemplifica quanto affermato dall'analisi.

Come sommario delle conclusioni raggiunte sui pattern, alla fine di ognuna delle due sezioni definite in precedenza, è presente una scheda di valutazione che, con lo stesso metodo utilizzato e definito dalla WPI, permette di avere una rappresentazione generale della realizzabilità dei pattern in JOLIE.

Il **quarto capitolo** riguarda gli esiti tratti dal lavoro di tesi, riportando un confronto tra le realizzazioni dei pattern in JOLIE e le valutazioni del WPI rispetto agli altri linguaggi da loro considerati e valutati. Sulla base di quanto ottenuto nel terzo capitolo vengono definite le conclusioni del lavoro portato avanti sui pattern e viene delineato un'eventuale scenario riguardante il proseguimento dell'opera concernente la validazione ed il completamento della studio.

In ultimo vengono tratte alcune conclusioni sia riguardo JOLIE, nel contesto evolutivo del linguaggio e soprattutto del progetto open-source che è alla sua base, sia sul SOC, considerato nell'ambito del BPA e del suo attuale ambito di sviluppo dinamico.

Summary

This work sets its focal point in the description, verification and demonstration of the feasibility of *Workflow Patterns* of *Control-Flow* and *Resource*s, defined by the *Workflow Patterns Initiative* (WPI) in JOLIE, an innovative service-oriented programming language developed within the context of *Service Oriented Computing*.

The Service Oriented Computing (SOC) is a new way of thinking about programming distributed applications, whose fundamental concepts are *services* and *composition*. The SOC approach defines the capability to build an application as a composition of services that realize its behavior, according to a particular workflow.

In order to provide the necessary knowledge for understanding theories, mechanisms and constructs of JOLIE used for the realization of patterns, the work of research conducted in this thesis has been divided into four parts, corresponding to the same number of chapters.

In the **first chapter** a general description SOC is provided, along with an introduction to Business Process Automation (BPA), which constitutes the environment where SOC is included. For this purpose a little historical overview about distributed systems has been included, up to nowadays. After that, the context of BPA is presented, including the innovations arising from its macro-components, whom SOC is part of.

Continuing the description of service oriented approach, its requirements (preconditions) are taken into account, after whom a precise definition of the term "service" is provided, which leads to the enunciation of SOC's principles, declined in the perspective of *Service Oriented Architectures*; ultimately the methods concerning service composition - *orchestration* and *choreography* - are presented.

The last section of the chapter considers SOC in a strictly industrial perspective,

highlighting its strategic features.

The **second chapter** focuses on the description of JOLIE, the fundamental aspects of service oriented approach, which deeply characterize its conceptual definition (SOCK), and the theory of composition of services.

This chapter isn't intended as a comprehensive description of all language's features, but gives an overview of theoretical concepts, data structures, operators and constructs of JOLIE used to demonstrate the feasibility of Workflow Patterns in the next chapter.

The **third chapter**, the longest and most central in the context of this work, concerns the implementation of Workflow Patterns in JOLIE. At the beginning of the chapter a description of the characteristics of WPI and Workflow Patterns is provided, furthermore, in the two macro-sections about *Control-Flow* and *Resource* patterns, there are some notions regarding the methods employed in patterns definition (e.g. the theory about *Colored Petri Nets*), as well as the conventions adopted by WPI.

Finally the main (experimental) work of this thesis is presented, which concerns the description of patterns, the analysis of their feasibility in JOLIE, along with a sample code that illustrates what emerged from the analysis.

As a summary of the conclusions drawn on patterns, an evaluation table at the end of each section is reported, using the same notation employed and defined by WPI to represent the feasibility of each pattern in JOLIE

The **fourth chapter** concerns the results drawn from this thesis work, showing a comparison between realizations of patterns in JOLIE and the evaluation of other languages made by WPI. Based on the outcomes of the third chapter, the conclusions of this work are drown, along with the delineation of a possible scenario, about continuing the validation and completion of this study.

Finally some conclusions are drawn regarding both JOLIE, in the perspective of the evolution of the language and especially the open-source project at its base, and SOC, considered within the context of BPA and its currently dynamic growth.

Contents

List of Figures

List of listings

Chapter 1

Service Oriented Computing

This chapter is intended as a general overview about *Service Oriented Computing* (SOC) and the environment in which its underpinning concepts have grown.

Before presenting the features offered by JOLIE and by its orientation to services and their composition, it's fundamental to understand *why* the next quantum leap in *Business Process Automation* (BPA), passes by *Service Oriented Computing*, distributed systems, and especially reliable and expressive technologies.

In the sections of this chapter are taken into account several topics, starting from a little historical introduction about BPA and SOC [1.1], continuing with a general overview about the four foundational concepts linked to BPA and its evolution [1.2].

After this general introduction the sections 1.3, 1.4, 1.5 and 1.5.1 are deeply focused in defining the elements and the main features of the Service Oriented approach, while the last section [1.6] tries to examine the features of SOC viewed as a new strategic asset for the enterprise.

1.1 A bit of history

Making a brief tour on the history of BPA, it's necessary to follow the evolution of information technology from the perspective of both distributed computing and information modeling.

The birth of information technology and BPA is linked to big-sized centralized processing machines, operated by one user at a time from simple terminals. *Terminals* were used as mere input/output devices, lot of data were processed by the *mainframe*, along with a considerable amount of computation time wasted by waiting the inputs of the single user, during his work session.

The evolution of this mainframe-terminal paradigm brought to information systems organized according to the (now) widely known *server-client* architecture. Servers, equipped with considerable computing power and storage, were dedicated to heavy-load general-purpose processing and data retention, while clients - running on personal computers - were used for processing specific and limited amount of data and for managing data input and output from the servers.

Next *peer-to-peer* (P2P) networking changed the concept of client-server paradigm, inasmuch as each peer of a P2P network works both as a server and as a client, thus enabling the distributed processing of data, among the whole network.

Finally the last generation of information systems relies on cooperative but autonomous, loosely-coupled and distributed components, which can be composed collectively, in a modular way, to provide new solutions.

The bleeding-edge of this approach to composition and collaboration, is brought further towards a higher degree of integration, not only letting each module cooperate with each other, but even to making them "*understand*" each other. This target is achieved by the fundamental division between data and application.

This first step, towards a more application-agnostic data management system, was made thanks to the creation of a standardized *Structured Query Language* (SQL) and the widely adoption of relational databases; although the semantics of data were still strictly linked to the ad-hoc solution coupled with the *Data Base Management System.*

The same kind of problem happened with the adoption of the Internet: for a long time, data published and retrievable on the *World Wide Web* (WWW) had no standard format nor standardized methods to access it. This characteristic did not compromise the incredible success of the WWW, since, in those "early" days, only humans were supposed to read, understand and process the information, whilst nowadays, it's becoming more and more important to automatically understand and process huge flows of information, possibly coming from different and not coordinated sources.

Thus, in recent times, the research for a "smarter" automated information processing lead to develop four main areas of innovation, that characterize the next generation of Business Automation technologies.

Anyway it's very important to keep in mind that the more independent and distributed the resources on the WWW become, the greater is their value, although this increased heterogeneity requires a greater effort to "extract" it. To this purpose, web services, by bridging the gap between heterogeneous systems, promise to ease the creation of distributed systems, whose high value can be seamlessly extracted.

1.2 A quantum leap in Business Process Automation

As aforementioned, BPA aims to develop technologies that make possible and (relatively) easy to extract value from distributed systems.

This technology is based on enabling the automatic creation of new services by combining already existing services and data between heterogeneous contexts, like business to consumers, business to business and internal business services.

The highly dynamic environment of today's businesses needs a real "quantum leap" in BPA, especially because of the closer and closer changes these systems are subjected to, which require a lot of adjustments and, mostly, the continuous verification of conformance between these changes and their working environment, that constitutes an unaffordable increasingly cost for companies.

The four areas of innovation that leads to this necessary leap are described in the next subsections.

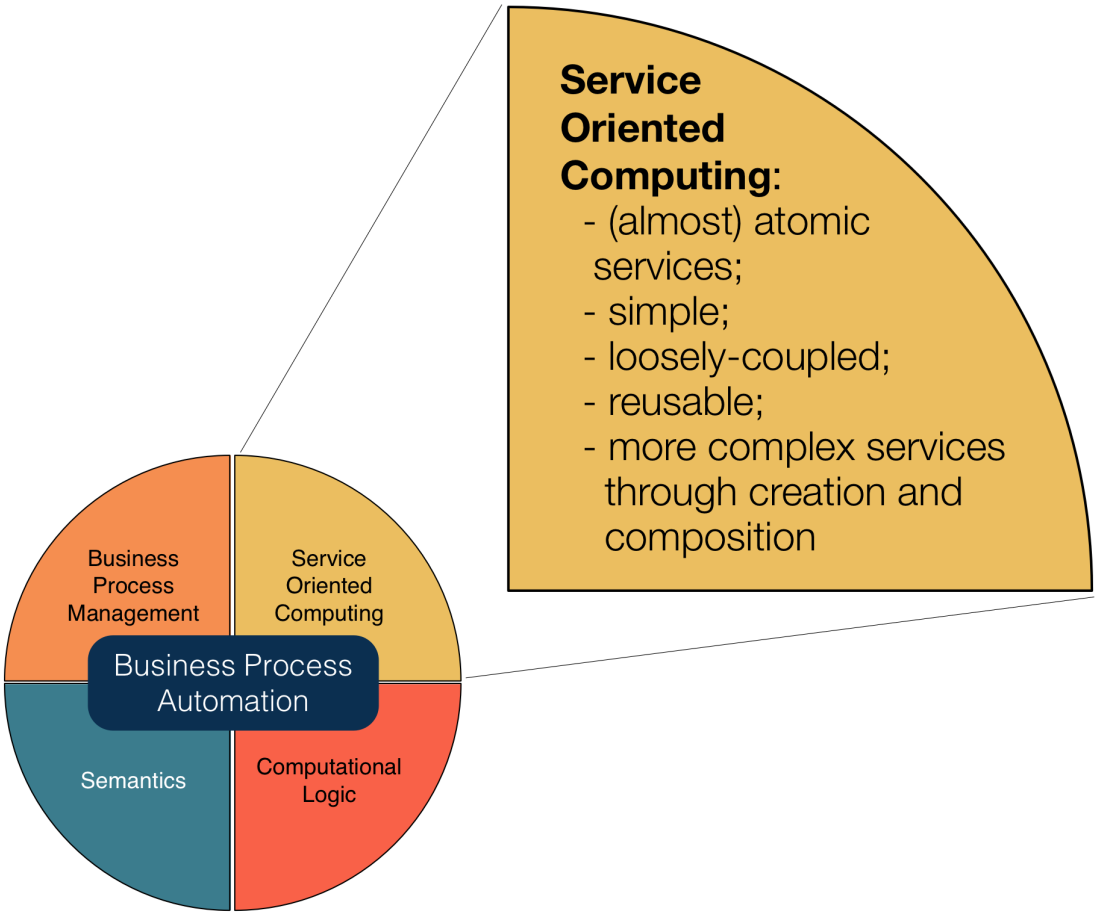


Figure 1.1: Business Process Automation Areas (SOC exploded)

1.2.1 Business Process Management

Business Process Management (BPM) is an approach towards management that considers it as a whole, whose focus is pointed at composing all the aspects of an organization by means of an abstract view of all of its components.

The practice of BPM is made by four phases about process:

- design;
- modeling;

- execution;
- monitoring and optimization.

One of the main technologies currently adopted by the market is BPEL – acronym of *Business Process Execution Language* – whose purpose is to formally describe business, helping to translate a design, which, generally, has been a "manual task" so far.

Furthermore the standard which defines how to employ BPEL in describing the interaction between (web) services is called WS-BPEL (Web Service-BPEL) and since its standardization, in April 2003, more than thirteen engines has been developed by companies like Oracle, Microsoft, IBM and SAP, in order to create a runtime environment for automatically created composition of services.

1.2.2 Computational Logic

Combination of preexisting services can give birth to new services, whose complexity may become more and more difficult to manage. That's why, another important area to push further the possibilities offered by BPA, is the automated composition of a variety of services, according to specific situations.

Computational Logic (CL) actually provides algorithms necessary to achieve this.

At the heart of CL is an application-independent inference procedure, which:

- accepts queries from users;
- accesses the "facts" in its knowledge base;
- draws appropriate conclusions.

Thus, together with semantic description of business contexts, it is possible to infer a sequence of activities and/or services that can solve a specific business goal. Furthermore, by reasoning on already acquired knowledge, CL may extract new information and record its conclusions in its own knowledge base.

This opportunity is reached by a conjunct work between Semantics [1.2.3] and CL, describing problems and knowledge through ontologies and reasoning with a wide set of algorithms and techniques like evidential, causal, induction, abduction, and deduction reasoning, along with natural language understanding and enhancing.

1.2.3 Semantics

Semantic models of business is the third pillar of BPA.

Semantics gives the power to express, in a machine-understandable language, the knowledge of a system as well as the characteristics of services.

Nowadays there isn't still a widely adopted standard in the strict sense, since the market offers a wide plethora of standards and formal specifications, but, towards this purpose, the W3C has proposed an architecture called the *Semantic Web Stack*, whose technologies shall be adopted as a standard for interaction between Semantic Web services.

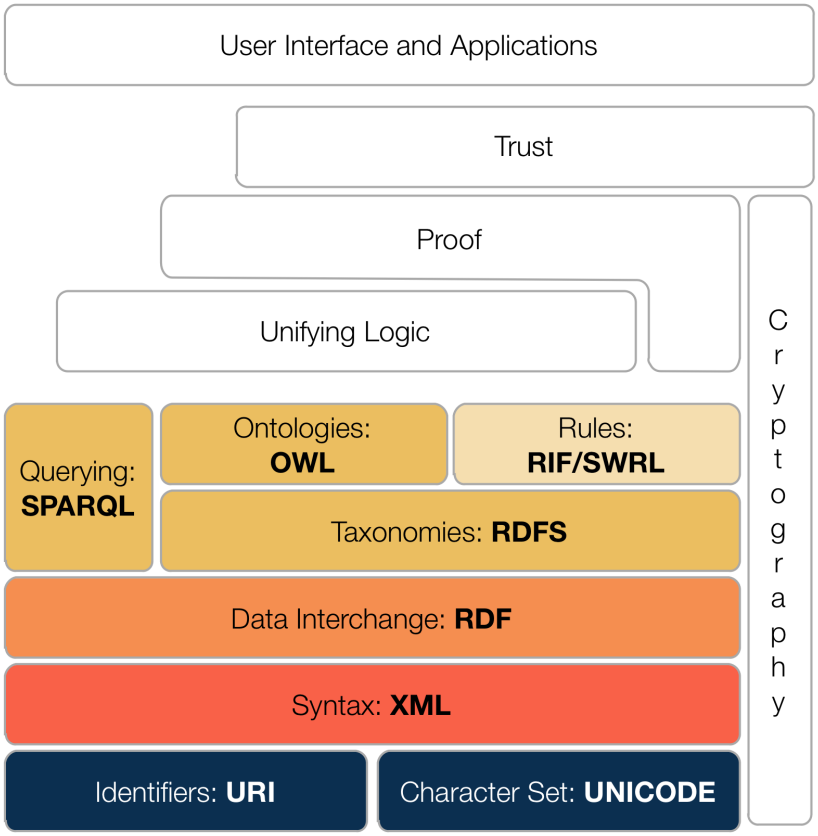


Figure 1.2: The Semantic Web Stack

Some of the protocol defined in the architecture have already been developed and

accepted as standard, like the *Resource Description Framework* (RDF), serialized into a wide range of interchange formats, like *XML*, *Turtle*, *N-Triples* and *JSON-LD*.

Along with RDF comes the *RDF Schema* (RDFS) and the *Ontology Web Language* (OWL) which provides respectively ontologies description and ontologies authoring, while the *SPARQL Protocol and RDF Query Language* (SPARQL) is the query language used to search through RDF-based data and statements.

In addition to the aforementioned technologies, some have to be still developed and standardized, like the *Rule Interchange Format* (RIF), whose alternative is the *Semantic Web Rule Language* (SWRL), and whose purpose is to support rules for describing relations, not directly described using the underlying technologies (e.g. OWL).

It's also notable that adequate methods and guidelines for annotation of business objects and services are currently a bleeding-edge research subject, as well as the tight group of tools available, and still in early developing phases, for working with semantics in a business environment.

1.2.4 Service Oriented Computing

The paradigm introduced with SOC is a relatively new approach in thinking and designing information systems.

The service oriented approach is pivoted in creating and composing loosely-coupled, reusable and simple (web) services, instead of focusing on the creation of monolithic, holistic and - in most cases - very complex applications, which bears with them all the drawbacks that concerns keep running, maintaining and upgrading such systems.

One of the focal points of this new way of thinking IT architectures, is the essential need for a language to make possible bind and compose services, defining either pre and post-conditions, along with a precise definition of procedures and their outputs.

Research in both academic and business worlds is in great excitement on the subject and it's currently addressing this issue, taking as reference web service technologies and in particular using WS-BPEL (a.k.a. BPEL4WS) [1.2.1] and WS-CDL (a.k.a CDL4WS) for programming compositions.

This is the context in which JOLIE comes into play, presenting an innovative alternative to WS-BPEL and in general to other orchestration languages as discussed in Chapter 2.

Figure 1.3 provides a brief layered view on the main technologies that characterize the stack of (web) services.

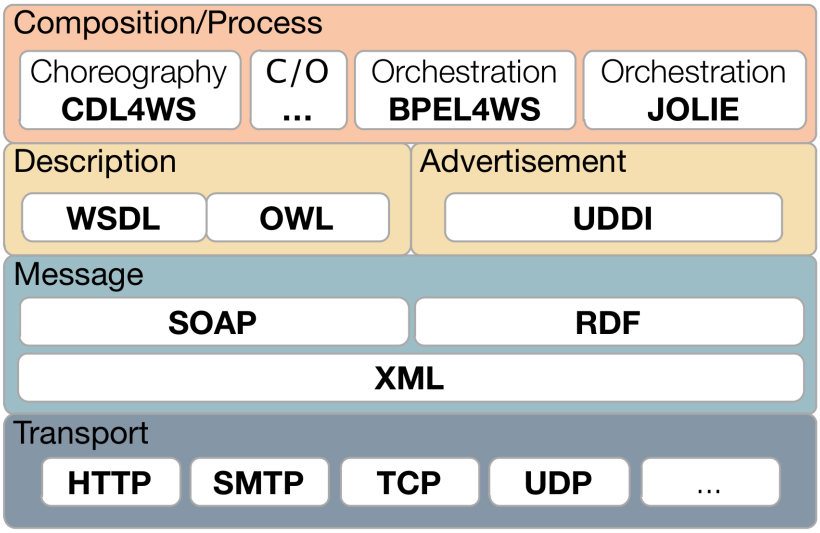


Figure 1.3: Web Services Stack (a view)

It's important to empathize that, while the current focus in research and development of new languages and tools is directed at defining a "static" approach to make possible composition of services with ease and precision by a programmer (human), the final endeavor of BPA and SOC is to allow services to understand each other, in order to find solutions to problems expressed by the conditions given by their users, letting them to automatically compose themselves.

1.3 Service Oriented Computing Preconditions

This section is intended as a continuation of section 1.2.4, focusing on SOC preconditions, i.e. all of those components that lead and are necessary for designing

and developing a service oriented system. Thus, let's begin stating that distribution, decoupling, composition and collaboration must necessarily live in "open environments".

The most straightforward example of this "openness" in the wide spreading of the Internet, instead of proprietary local area networks, which has been achieved only with the adoption of a *open* approach towards heterogeneous and autonomous systems. The key-concepts behind such an idea of openness are autonomy, heterogeneity and dynamism.

1.3.1 Autonomy

The autonomy of services reflects the autonomy of businesses and people behind them.

Each organization which builds an application must be endowed with the capacity to autonomously implement and control its own services, in the same way as these organization are autonomous but still integrated into a - usually - thick network of relations and transactions.

Furthermore autonomy is the leading path towards security and robustness, since each service must be designed as atomic and not affectable by other services with which it cooperates; this fact is very important considering that third party services may have been implemented by misinterpreting guidelines or with faults.

It's also true that autonomy means the impossibility to control (to force) other (third party) components to do any specific task. This is the most explicit case of robust service design, such that, if the requested task isn't completed, the requesting application can "understand" it and apply the corresponding countermeasures.

1.3.2 Heterogeneity

Along with autonomy goes heterogeneity.

Heterogeneity means no centralized or top-down design over the different components of a system. This approach toward services has both historical and "sociopolitical" reasons, since the most part of today's organizations have a good

chance to have accumulated a substantial set of legacy applications, components and business objects which, at the time of their design, were built for mostly single or narrow purposes. Since "collaboration" wasn't among requirements, each designer made his own application based on his knowledge and suiting to its working environment, along with needed performances and constraints linked to its tasks.

Thus applications heterogeneity can be declined at each level of a system, starting from transmission media, to networking protocol, to encoding and serialization of information and data formats.

On this matter it's remarkable that, while maintaining an open approach, standardization of protocols and formats (not design) can improve productivity by means of enhanced interoperability[1]; it's also notable that composition along with migration, data retrieval and conversion from legacy systems is considerably simpler and less error-prone if supported by the employment of standards, especially the open ones.

1.3.3 Dynamism

Summed together heterogeneity and autonomy lead to dynamism among system which collaborate into an open environment.

Dynamism is the essence of SOC: any environment in which services and applications aren't controlled directly by a central manager and whose implementation is not pre-determined, is a dynamic one.

In this environment services can both change their behavior according to their owners' needs and become available or unreachable independently.

Designing services that collaborate in such a system means taking into account and dealing with this dynamic behavior, in which components can be available, depart, be modified and substituted dynamically.

[1] *Internet Protocol* (IP), *HyperText Transfer Protocol* (HTTP), *Universal Characters Set* (UCS) and *UCS Transformation Format*s (e.g. UTF-8), *eXtensible Markup Language* (XML) are some of the most known widely adopted standards in the IT world.

1.4 What a "*service*" is

Defining what can be identified a "service" has the same burden that theorists and technologists of previous generation tackled in defining the concept of "object", during the development of the object oriented paradigm.

Different organization gives different definition of service, some of them are reported as it follows:

> *"A piece of business logic accessible via the Internet using open standard."*
>
> *Microsoft*

> *"Services are collections of capabilities.*
>
> *A Service is a unit of solution logic to which service-orientation has been applied to a meaningful extent.*
>
> *Services exist as physically independent software programs with specific design characteristics that support the attainment of the strategic goals associated with service-oriented computing.*
>
> *A Web service is a body of solution logic that provides a physically decoupled technical contract consisting of a WSDL definition and one or more XML Schema definitions and possible WS-Policy expressions.*
>
> *In a Web service Capabilities are exposed as operations."*
>
> *Thomas Erl*

> *"Loosely coupled software components that interact with one another dynamically via standard Internet technologies"*
>
> *Gartner*

> *"A software application identified by a URI, whose interface and binding are capable of being defined, described and discovered by XML artifacts and supports direct interactions with other software applications using XML-based messages via Internet-based protocols."*

World Wide Web Consortium

It's worth reading and understanding how difficult can be to converge towards a single definition, while each organization and person tries to figure out, according to their different perspectives, backgrounds and concerns, the meaning of a service and, on top of that, the possibilities of a still-nebulous matter such as SOC.

The least common denominator of all of those definitions is that a service is a capability provided and exploitable, either remotely or not. According to this viewpoint, the definition of a Web Service is a service whose functionality can be required over the Web.

Stated the basic definition of what a service is, the next section tries to delineate the main principles of Service Oriented Computing.

1.5 Principles of Service Oriented Computing

SOC represents a new paradigm in thinking, designing and building business applications which focuses on openness and flexibility, that, in turn, mirrors itself into a new way of thinking about business work items, offered service and structures.

1.5.1 Service Oriented Architectures

Switching to a new design paradigm necessitate the switch to a new architectural paradigm, such that the way the system is structured reflects the same essential properties its components share. In this way the whole system can count on a solid structural and theoretical integrity, which fosters integration, collaboration and development among services.

This kind of approach is called *Service Oriented Architecture* (SOA) and its main aim is to employ the principles of SOC into a cohesive framework - the architecture itself - in order to be applied seamlessly in software development and production.

Since this research area is receiving so much attention from either the scientific and the industrial community, several "SOAs" have been provided from multiple sources, each satisfying the underpinning elements of SOC. Thus no particular

SOA will be taken into account in this work, instead the key elements of SOA - and therefore of SOC - are listed and described.

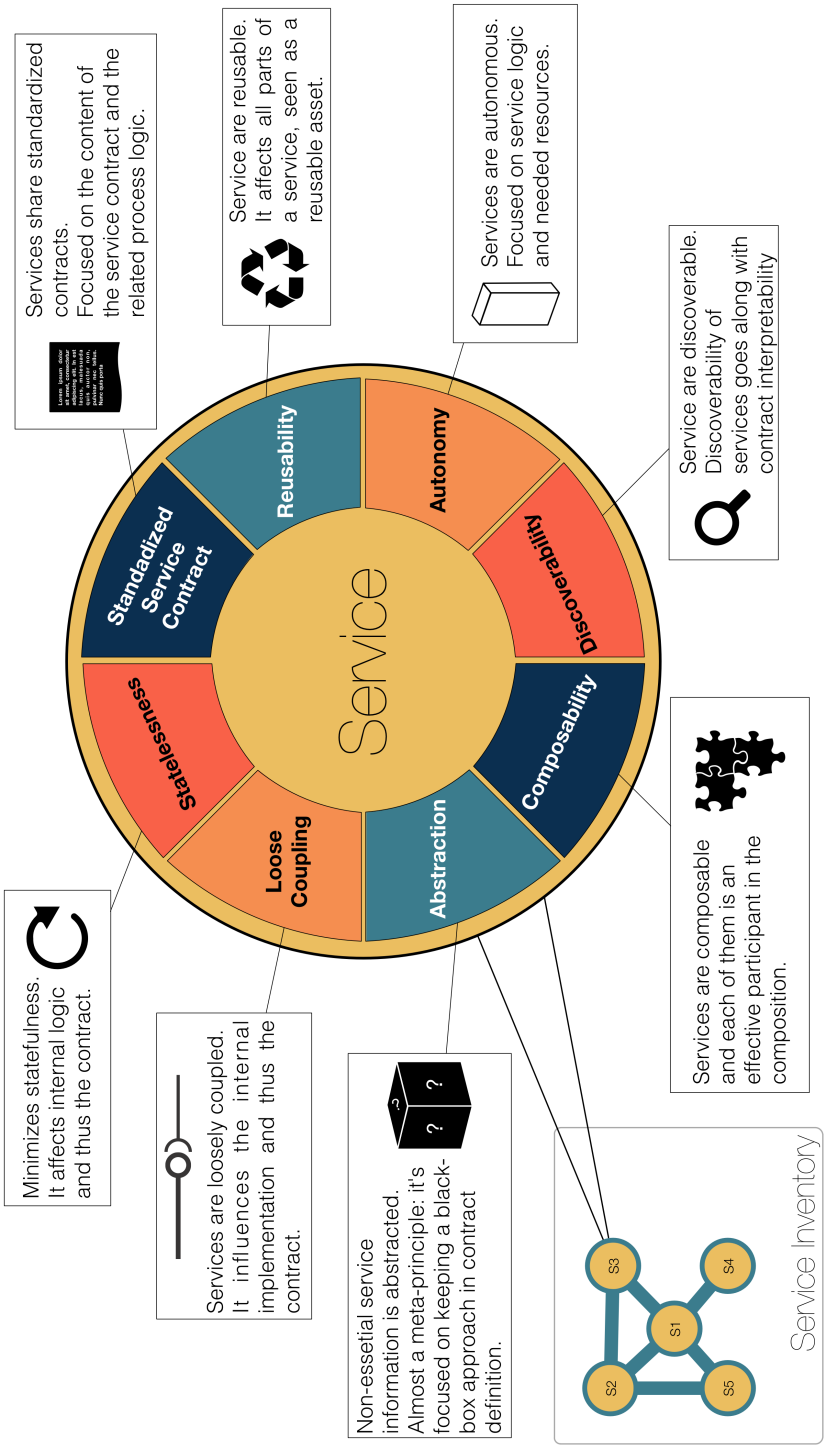


Figure 1.4: Principles of Service Oriented Architecture

1.5.1.1 Loose Coupling

Services are loosely coupled, thus they implement low consumer coupling requirements and are themselves decoupled from their surrounding environment.

By fostering a reduction in coupling between and within services, their contracts became increasingly independent from each other. Such independence is guaranteed by their contracts too, lowering the dependence between a service and its implementation.

In this way service can seamlessly evolve, lowering the impact of their evolution within their consumers environment.

1.5.1.2 Service Contract

Service within the same service-inventory are in compliance with the same contract design standard.

This means that a service contract - the collectively defined communication agreement between services of the same service-inventory - shall both be provided with the service and standardized by a general application of design standards. This element is important to achieve a meaningful level of interoperability either between services from the same inventory or from different ones.

Compliance on contracts leads to consistency between data models and consequently to a general simplification in understanding purposes and capabilities of services.

1.5.1.3 Abstraction

Non-essential service information is abstracted.

This principle states the importance to keep contracts light, publishing only essential information about the service, in order to provide a concise and balanced amount of details about the service, preventing unnecessary access to additional non-essential details.

In this way services guarantees a high level of abstraction from logic, functions and technology of its implementation, while proving the essential information about constraints and requirement necessary to interact with them.

1.5.1.4 Reusability

Services are reusable.

The SOC way to think about services is also about considering them like assets. In this perspective, reusability of assets is one of the highest strategic key point: using an acquired asset for multiple tasks increases the return on its initial investment.

Furthermore the reusability of services may give birth to a "service inventory" - or even to a set of them - which contains services whose logic is associated with a sufficiently agnostic context, to be reusable in any usage scenario.

1.5.1.5 Autonomy

Services are autonomous.

Services are highly autonomous in determining and controlling their underlying runtime execution environment.

This principle requires an high level of control over how service logic is designed and developed and above all, it must be well-defined the functional boundary of a service, which, as the principles states, mustn't be overlapped with the one of another service.

Thus execution environments of services must foster service-exclusive level of control, high concurrency and distributable deployments.

1.5.1.6 Statelessness

Services must minimize statefulness.

Statefulness means low degree of scalability in order to maintain a permanent state. In the contrary, services' states must be temporary and develop a state-agnostic logic, with state management deference.

State deferral is the key-concept of this principle, since by deferring the management of state information, consumption of resources is lowered and the potential reuse of the service is improved.

1.5.1.7 Discoverability

Services are discoverable.

Services communicate meta-data by with they can actually be discovered and interpreted.

An organization which possesses a large - or a lot of - service inventory, can benefit of an high level of discoverability of its services, each service's purposes and capabilities are clearly expressed, thus both humans and software programs can understand and interpret them.

To achieve an high level of service discoverability the existence of design standards, governing the meta-data used to make service contracts discoverable and interpretable, is imperative.

1.5.1.8 Composability

Services are composable.

Although service composability can be seen as a declination of service reusability - which fosters and enables wide-scale service composition -, composability brings this concept a step further, defining one of the most important and ultimate goal of SOC.

Composability of services means to consider each service as a potential member of a composition and thus, approaching to its design and realization with this capability in mind.

Composability encloses all of the aforementioned principles since, to achieve high efficiency in composition among services, reusability must walk along with highly efficient execution environments, in which concurrency is boosted at its maximum degree, managing resources used by services in the most efficient way - statefulness and autonomy - and designing flexible contracts in order to facilitate different types of data exchange requirements for similar functions.

1.5.2 Composing Services

In the previous sections the main principles of SOA and SOC have been listed and commented, from that exposition emerged that the criterion of composability

among services is the all-comprehensive key-concept.

Although the definition of the principle of composition states the underpinning requirements in designing services and their environment, it's noteworthy that, like any other abstract principle, it asserts the free interpretation of the modalities in which the composition among services is realized.

Focusing on this topic, two opposite approach can be followed in realizing composition of services. These two techniques face the problem of composing services from two perspectives at odds: one from a *choreographic* (distributed) point of view, the other from an *orchestrational* (centralized) one.

1.5.2.1 Choreography

Choreography focus on the collaboration of each party in the composition. Any service that takes part in the interaction knows and describes its own role (or part), while Choreography tracks the sequence of messages sent and received by the participants. Thus Choreography deals with the public message exchanges that occur between multiple services without defining a "director" of the conversation and simply defining the "contents" of the conversation that should be undertaken by the participants, letting each party collaborate in a peer-to-peer interaction.

Hence Choreography is about describing the interaction between each service, which comprehends control-flow[2], data-flow, message correlation (sessions), time constraints, transactional dependencies and the like; these elements are needed to capture and control interactions from a global perspective, letting each peer play its own role.

1.5.2.2 Orchestration

The second approach for composing services is to orchestrate them. Orchestration looks at the problem of defining the collaboration among services from a centralization perspective.

The orchestrator is a "superior" process (service) endowed with the capability to regulate communications and execute internal actions including data transformation, and advanced composing operations - not declared in any other service

[2]not on the internal logic of a service, but on "correctness" of the sequence of interactions between services.

interface, like embedding other services or legacy applications for both management granularity of resources and system upgrade with backward compatibility.

Hence service composition, in the context in a Orchestration, may make use of operations which overcome the *simple composition*, allowing the orchestrator to perform advanced compose operations. In Figure 1.5 the *simple composition* of services is represented by the interactions with *Service C* that are performed over the network.

As a foretaste of JOLIE overview provided in Chapter 2, three of these "advanced compose operations", taken from the operational context of the language, are analyzed.

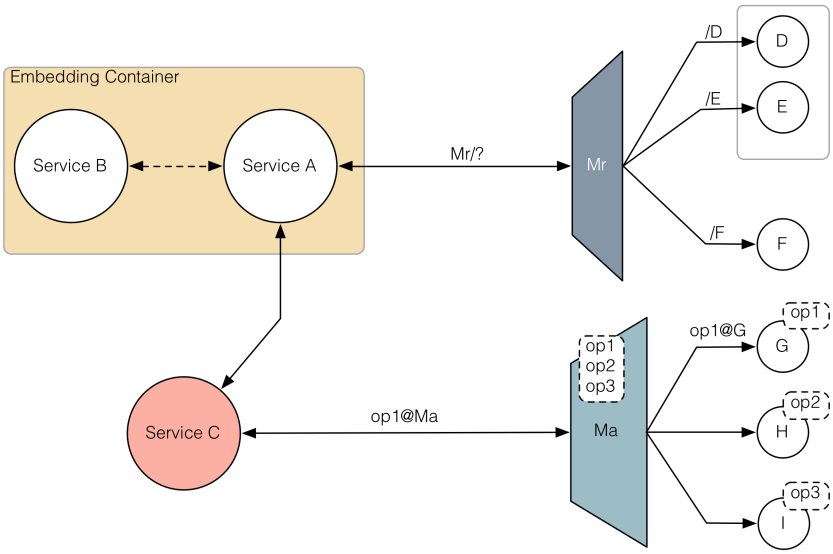


Figure 1.5: Orchestration Service Compositions

1.5.2.2.1 Embedding

> *Embedding is the composition of more than one service into the same container.*

Embedding services into the same container allows the use of internal communication mechanism without the need of the network. This feature can be exploited for security, backward compatibility (legacy systems) and performance issues.

Furthermore the capability to embed services boosts the granularity management of services in the systems, as well as making possible to overcome limitations and performance degradations that comes with passing each communication via the network.

On the contrary, the practice of embedding "utility" services like the ones providing mathematical functions, logging or time management unburdens the need to deploy each auxiliary service in the network.

In Figure 1.5 the embedding of services is represented by the inner interaction between services A and B.

1.5.2.2.2 Redirecting

> *Redirecting allows for the creation of a master service, acting as a single communication endpoint to multiple services which are called resources.*

A redirecting service - the service *Mr* in Figure 1.5 - is a *master service* which receives messages directed to a certain resource (service) and then forward them towards the corresponding services.

This behavior is obtained by binding an input port of the master service to multiple output ports, each identifying a specific service (via its "resource" name).

This approach give birth to a wide range of architectural possibilities, like the capability to provide an unique gateway for the clients of the system.

In this way the dynamic relocation/replacement of services can be done transparently with respect to clients; furthermore this capability can be exploited for "filtering" and even "transforming" messages directed towards the resources, making possible for the master to act like a router/firewall, forwarding, commuting or discarding incoming messages from outside the system.

It's worth noting that redirection not necessarily hides the services behind it, as represented in Figure 1.5 where service A can invoke any of the redirected resource by means of their names (anyway passing through *Mr*).

1.5.2.2.3 Aggregation

Aggregation is a redirecting composition of services whose interfaces are joined together and published as unique.

Aggregation can be seen as an evolution of the redirection service.

In this case the master service groups more services under the same interface, thus no "resource" is directly callable by an outer communication. On the contrary the master service exposes an interfaces which declares to provide the functionality of the resources it aggregates.

The concept of this kind of composition that, instead of simply redirecting to services, aggregation makes clients loose any knowledge about the services behind the aggregation itself. Hence, one of the main advantages of this composing method is to deal with the necessity to completely hide the component of the system from the outside.

With reference to Figure 1.5, *Ma* represents the aggregation service which receives the request to execute *op1* from an outer service (C) and seamlessly passes it to service G which implements it.

1.6 In the shoes of an entrepreneur

Service orientation it's being a long-talked topic in both research and business environments as part of the forthcoming BPA revolution treated in section 1.2. In previous sections SOC principles and its approach have been diffusely described and commented, but it's worth to keep in mind that, besides the importance of a solid and consistent theoretical structure, the affirmation of a new technology or, in this case, of a new way of thinking about applications and their architectures, is strictly linked to the adoption of that technology in business world.

Considering SOC from the perspective of an entrepreneur - and more generally, from an industrial point of view -, gives the opportunity to cross the wall which usually divides research departments and industrial production chain, highlighting strengths and weaknesses of the "new kid on the block".

1.6.1 Long-lasting, easy changing

As stated when enumerating the principles of SOC, thinking about applications, in a service oriented context, means considering a long-time, profound and durable change in application architectures. In particular, the major modification in the way an application or, generally, a software asset shall be seen, regards the change itself.

Until recently, it has been the degree of adaptation of a design model (or an architecture), with respect to the business needs, the main key-point for deciding on which technology to invest in.

Contrariwise, the fast-changing environment in which the most part of industries are embedded nowadays, requires a reconsideration, fostering the degree of adaptability in place of adaptation.

This is a real revolution in thinking about business assets and industrial planning, since applications have usually been considered long-lasting - and dedicated - investments, whose biggest span in their lifecycle begins with their release (deployment), as the final output of a development process. This span is formally called "maintenance" and it's characterized by an intensive use of the application, which, at most, may require the implementation of minor changes, bugs fixing and performance adjustments. These are the only interventions operated on the software until its removal.

Conversely the present and near-future strategic value on which basing the evaluation of application designs and architectures is their degree of adaptability towards new, complex and systemic needs.

In this context SOC plays an important role, since thinking applications in term of services is thinking about them in the terms of their relations, their connections and the way they can be changed, combined and composed, in order to achieve either general or specific solutions.

The modular and additive approach of SOC fosters the accumulation of functionality which, from a restricted initial set, can rapidly expand by building and adding new services on top of the previously acquired ones.

1.6.2 Enterprise Integration

Orientation to services isn't a new concept for thinking about resources in a firm: the most part of business functions in modern enterprises are viewed as services and the resulting organization model fosters the aggregation of staff and resources according to the services they perform within the organization itself. Each department's services are needed by and depend from others, in order to fulfill their tasks and thus the objectives of their company.

The aforementioned concept applies for designing service oriented applications, which are characterized by the principles of standardization, abstraction, reusability, autonomy and composability described in section 1.5.

Actually there's much ado about enterprise integration of SOC and, as any evolving and not clearly defined technology, its principles - and its buzzword in particular - has been applied, to some extent, to a wide set of applications from any kind of distributed system, even including process, policy or semantic management systems.

As clearly stated in section 1.2, there is a subtle but important difference between all of these concepts, and to fully take advantage of them, it's important to understand their role, their tasks and even the "gray" areas in which they can overlap in a synergistic manner.

1.6.3 When it's good to go SOC

Competition, globalization and faster technology advances call for fast-adapting companies, whose capability to conform their structure to new and unexpected market conditions determines their performances and, ultimately, their success.

The main benefit of SOC is that, by building applications in a modular and composable way, efforts and time needed to integrate a new functionality in the enterprise system drastically decrease.

However, like each new and raising technology, SOC isn't the silver-bullet for taming any problem of an enterprise, rather it might become an issue if applied thoughtlessly.

SOC is a technology which requires an high level of acquaintance about either its principles and the objectives to be accomplished, along with an ad-hoc process

plan to coordinate design of interfaces and development of modules.

However, while getting acquainted with SOC requires some significant initial investment, these expenses rapidly bear their fruits and a lot of companies that have started using SOC continue and expand their use of it.

Clearly shifting from a "monolithic" to a service oriented approach bears some issues, above all linked to re-thinking from scratches the previously realized applications in term of their services, trying to boost each component's reusability.

Compared with traditional and even client-server applications, SOC systems introduces the need to address a range of theoretical and practical issues, derived from the necessity of managing the interactions between applications, spread across multiple platforms, possibly running on an heterogeneous mix of operative systems and developed by separate and autonomous groups.

SOA is part of the solution, since its principles clarifies the system design, isolating each module from each other and boosting the creation of a efficient documentation of interfaces.

Testing the adaptability of SOC in the specific context of a company is relatively easy and bears a little cost, but choosing to completely "go SOC", it's a strategic choice leading to significant long-term costs (and benefits), which must be well pondered, as it commits a deep impact in how business units relate to each other.

1.6.4 Never change a winning team (Legacy Systems)

Today's companies have accumulated a lot of assets in terms of data and applications - procedures - which constitutes both a strategic resource of success and a damper to enterprise's business modernization.

Handling preservation, conversion and integration of legacy systems has a strong economic impact on the budget of any company.

Each system (application or data) acquisition has required an investment by the company and returns a profit. Hence the firm benefits of this investment until its returns are overcome by expenses connected to its maintenance, inefficiencies or lock-in switching costs.

While switching to a new architectural paradigm usually means undertaking new costs linked to legacy systems replacement, using SOC for designing an "um-

brella" system over the whole company, allows the achievement of two features: retrofit of legacy systems and sharability of legacy systems' services.

By retrofitting a legacy system into a larger, service oriented, one, the investment undertaken for its acquisition, isn't "lost" in (re)building a new one - possibly with the same identical features - but it becomes part of the development of the new system, where some (selected, useful) parts of the legacy application are integrated as services.

Furthermore the newly retrofitted services become usable by (and shared to) any other service or business unit, letting the whole organization take advantage of the services provided by the legacy system.

Chapter 2

A Java Orchestration Language Interpreter Engine

The following chapter is focused on the description of JOLIE, the fundamental aspects of the service oriented approach which deeply characterize it, and the theory of services composition.

This chapter contains both theoretical definitions and practical examples, but it's noteworthy that the following sections are not intended to be an exhaustive description of all of the features, constructs and theoretical concepts behind JOLIE, instead it's designed as a brief overview of the language's constructs used in this work, in order to provide the required knowledge to understand the realizations of workflow patterns given in Chapter 3.

Such overview is divided into two sub-parts: the first traces a path that starts with the definition of operations and communications among services, leading to their composition and instances management; the latter relates to "side" concepts of the language, which are involved in data and workflow handling.

As an introducing example, Figure 2.1 provides the representation of an interesting composition of services. In the example Service 1 is the master service (orchestrator) of the composition which manages separately the communications with other services by means of their sessions. These communications are made on various media and protocols. Service 3 communicate via Bluetooth using the SOAP protocol, Service 4 via the Internet (HTTP/TCP/IP), while Service 2, using the internal network which also Service 1 belongs to, embeds a legacy system by

means of a Java Bridging Class.

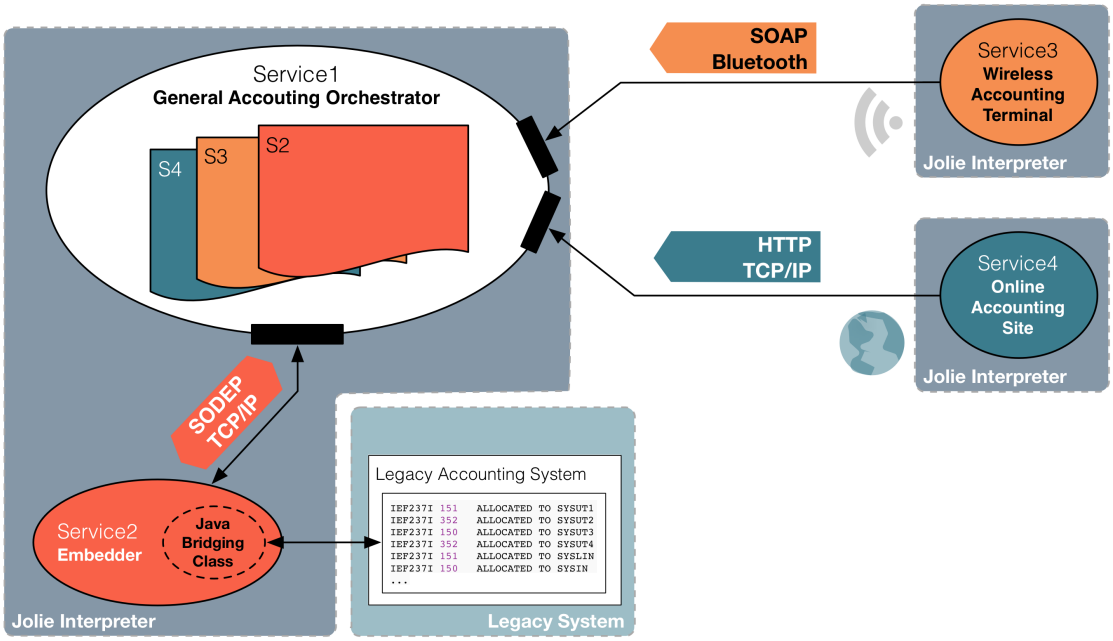


Figure 2.1: A composition of services in JOLIE

2.1 Dealing with services

2.1.1 SOCK & JOLIE

The fundamental feature of JOLIE language is its innovative approach in composing services.

While maintaining an imperative programming paradigm - which contrasts with the declarative one, used by the most part of other orchestration languages - JOLIE provides a simple yet expressive environment in which services can be composed in several different manners.

JOLIE has been developed as a realization of the theoretical process calculus *Service Oriented Calculus Kernel* (SOCK) which is a process calculus developed for representing service behavior, engines and systems.

Organized in five layers (*service behavior*, *service engine state*, *service engine correlation*, *service engine execution* and *services system*), one overlapped on the other, SOCK is meant for precisely describe each possible execution path of a composition of services, whose behavior is the emergence of actions and executions among the level previously defined.

In SOCK are discussed the main features of communications, based on external/internal inputs, output actions and service locations, that are the theoretical basis upon which the JOLIE language has been built on.

The primitives defined by SOCK and involved in modeling such behaviors are four and are divided in *input* and *output operations*.

Input operations are:

- *One-Way*: which has the task to receive a request message;
- *Request-Response*: whose task is to receive a request message and to send back a response message to the invoker.

Output operations are:

- *Notification*: that has the task to send a request message (complementary action of the *One-Way* operation);
- *Solicit-Response*: whose task is to send a request message, waiting for the corresponding response to be sent back from the invoked resource (complementary action of the *Request-Response* operation).

Based on this approach, JOLIE implements orchestration between services upon their communication.

2.1.2 Services and Operations

Given the approach described by SOCK in the previous section and the fact that JOLIE is strictly related to this concepts, the practice of writing a program in JOLIE results in the composition two parts: a *behavioral* and a *deployment* one.

As a matter of facts, the *behavioral* part is about the definition of the workflow of the orchestrator, which directly corresponds to the higher levels of SOCK's stack, whereas *deployment* part relates to the lower (*engine*) levels of the stack.

Mention should be made apart for the *service system* layer, which has no direct correspondence in JOLIE language, although its definition finds its realization in the *Communication Core* of the language interpreter.

Services can invoke (and in a complementary manner, being invoked by) other services, in order to make possible a more complex behavior, towards the completion of a defined task. Composing services is all about making them communicate with each other, and communications in JOLIE are modeled upon *operations* identified by SOCK and reported in Section 2.1.1.

Declined in JOLIE context, an *operation* is defined as a functionality which is exposed by a service and that can be invoked by other services. As already stated, there are two different kind of input operations, namely *One-Way*s and *Request-Response*s, to which correspond their output counterparts called *Notification*s and *Solicit-Response*s.

Let's consider the scenario of two services, namely the *requester* and the *responder*, in which the requester wants to invoke one the services exposed by the responder. The invocation of one of responder's services by the requester implies, at first, the definition of an *output port* in the requester's preamble. Such definition is required to state *Location*, *Protocol* and *Interfaces* (*operations*) exposed by the responder and that are invoked by the requester itself.

This is made possible in JOLIE by the `outputPort` statement, whose example is shown below:

Listing 1: outputPort statement

```
1 outputPort Responder{
2     Location: "socket://localhost:8000"
3     Protocol: sodep
4     Interfaces: responderInterface
5 }
```

According to the example above, the output port identifies:

- the location (of the responder's service) at the localhost of the running system (at port 8000);

- the protocol used to communicate is the *Simple Operation Data Exchange Protocol* (SODEP);
- the operations that can be invoked are described by the *myServiceInterface* interface (whose definition is discussed afterwards [2.1.3]).

Once defined the invocation of the service, it must be declared the corresponding input port in the responder's preamble; specular to the outputPort, the `inputPort` statement example is provided as it follows.

Listing 2: inputPort statement

```
1 inputPort Responder{
2     Location: "socket://localhost:8000"
3     Protocol: sodep
4     Interfaces: responderInterface
5 }
```

It's worth noting that JOLIE has been expressly designed with a transport-and-protocol-agnostic approach, in fact by specifying Locations and Protocols of an interface, any kind of communication medium (TCP/IP, Bluetooth, Java RMI, etc.) and protocol (SODEP, SOAP, HTTP, etc.) can be used to let services communicate to each other.

2.1.3 Interfaces

In the previous section has been taken into account the definition of the "rules" (location, protocol to use, exposed interfaces) of communication between services.

This section is focused on delineating the meaning of a service *interface* and its purposes in JOLIE.

An *interface* is a collection of operations that can be used (and re-used) in input and output port definitions. In JOLIE interfaces are a very useful means of sharing services' interfaces between different applications. With reference to the example in previous section, there's no need to duplicate the definition of the operations invoked by the requester and exposed by the responder; on the contrary, it suffices to declare the *respoderInterface* once, by means of the `interface` construct.

In the scope of the interface definition, OneWay and Request-Response operations can be specified by the corresponding statement (`OneWay:` and `Request-Response:` respectively) followed by their names, one after the other, divided by a “,”.

This is done, in practice, by writing a `responderInterface.iol` file, whose content is reported as it follows.

Listing 3: The inputPort statement

```
interface responderInterface{
    OneWay: oneWayA, oneWayB
    Request-Response: request-responseA, request-responseB
}
```

Finally the interface can be `included` in both requester and responder applications (written in `requester.ol` and `responder.ol` files) by means of the `include` statement.

For the sake of brevity, only the code of the requester is showed below.

Listing 4: requester Code

```
include "responderInterface.iol"

outputPort Responder{
    Location: "socket://localhost:8000"
    Protocol: sodep
    Interfaces: responderInterface
}
```

As can be seen, the responder exposes (and the requester can invoke) four operations, two One-Ways and two Request-Responses. A graphical representation of the structure assembled until here is given by Figure 2.2.

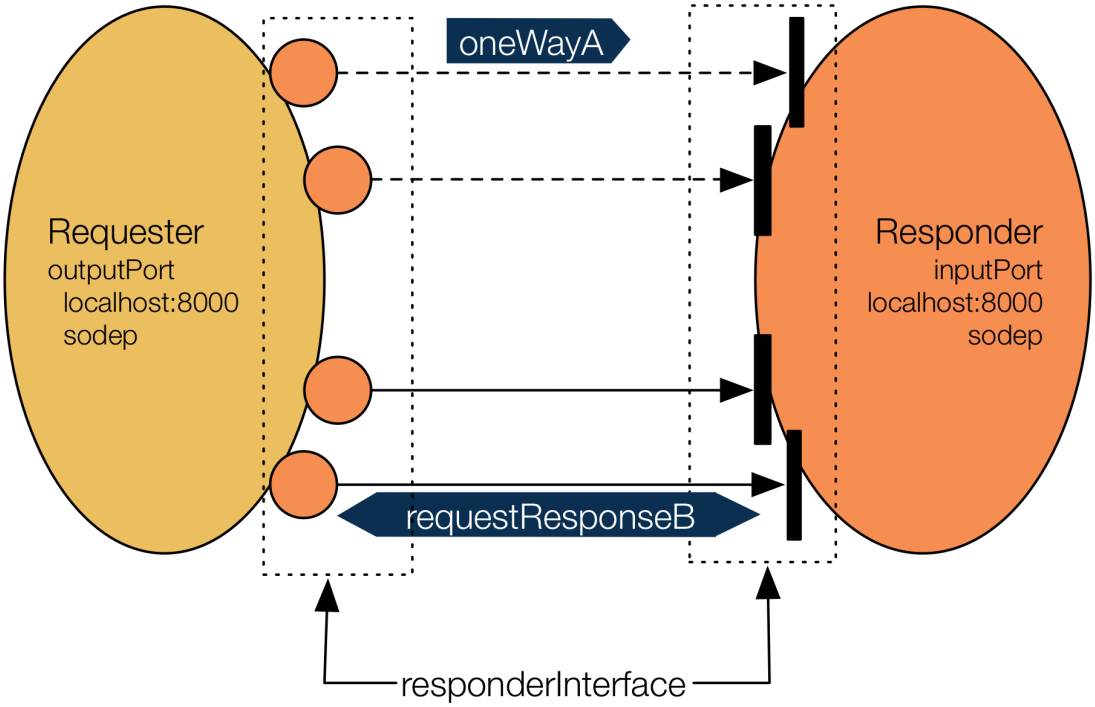


Figure 2.2: Requester-Responder Structure

2.1.4 Composition of statements and services

Once described how services can communicate with each other, it's important to understand how services' behaviors are realized in JOLIE.

Each JOLIE program must define a `main` procedure, which represents the service's entry point of execution. The `main` procedure can contain any kind of process and moreover, it can be preceded (or succeeded) by *definitions* of auxiliary procedures and/or *initialization* code[1].

As aforementioned, JOLIE offers a *definition* statement (`define`) which allows the specification of procedures callable from other code, by their name. It's noteworthy that, defined procedures do not implement any "procedural programming" behavior (value passing, local variable state and the like), and their implementation in JOLIE is only meant for recurrent code repetition avoidance. For a better

[1]the initialization (init) block is explicitly described in section 2.1.5.2

understanding of defined procedures and data handling, is made reference to the Section 2.2 about data structures in JOLIE.

Composition rules and statements are the same for either `main`, `init` and defined procedures and they offers the possibility to compose statements in sequence, parallels or in a non-deterministic input choice manner.

2.1.4.1 Sequence operator

The sequence operator in JOLIE is ";"[2] and it specifies that the statement at its left side is executed before the one at its right. For a more accurate description of this operator is made reference to the section 3.2.3.1.

2.1.4.2 Parallel operator

The parallel operator in JOLIE is "|" and it specifies that both statements which surround it are executed concurrently. For a more accurate description of this operator is made reference to the section 3.2.3.2.

2.1.4.3 Mixing Parallel and Sequential composition

It's important to understand the priorities set in JOLIE when mixing sequentially and parallelly composed statements.

In fact the parallel composition has always priority over the sequential one.

Thus, considering an example in which are involved four statements and whose composition shall be:

- the execution of A;
- followed by the parallel execution of B and C and, after their completion;
- the execution of D.

[2] not be confused with the symbol frequently used by other languages as *statement termination* operator (like C, Java, Perl, etc.)

We could be induced in thinking that such an execution could correspond to the code example given below.

```
1 main{
2 A ; B | C ; D
3 }
```

Conversely the given code defines the parallel execution of two branches: the one corresponding to the sequential composition between A and B and the other one between C and D.

Such behavior is easier to understand if curly braces are used to enclose sequential behaviors, as shown as it follows.

```
1 main{
2 {A ; B} | {C ; D}
3 }
```

For the sake of completeness, the sought behavior defined at the beginning of this example can be obtained running the code given below.

```
1 main{
2 A ; {B | C} ; D
3 }
```

2.1.4.4 Non-deterministic input choice

The non-deterministic input choice construct is used to allow the programming of input guarded choices over a set of input operations.

Once an input operation is received, all of the other (waiting) input operations are discarded. To each input operation corresponds a branch of code, which is the only one that is executed.

The definition of a non-deterministic input choice block is expressed within square braces and it's followed by the corresponding code block contained between curly braces. For a more accurate description of this operator is made reference to the section 3.2.3.4.

It's noteworthy that a non-deterministic input choice blocks the whole execution of the program it's located in, until one of the programmed input operation is triggered by the reception of a message.

2.1.5 Sessions

JOLIE provides a powerful mechanism for session generation and management. In general each JOLIE service is also a session manager and a new session is initiated as soon as the first input operation is invoked. It's important to understand that, in JOLIE, each "instance" of execution run the same code but with its own (private) data.

JOLIE defines three kind of session executions:

- *single* (set by default): the execution is set as a "one-shot" instance. No other instances can be invoked after the first one;
- *concurrent*: each invocation spawns a new session which is run concurrently with others;
- *sequential*: each invocation spawns a new session, each of which is executed one after the other, ordered according to their time of invocation.

The JOLIE statement used to specify the session execution is `execution`, it's declared in the service's preamble and its argument, enclosed within curly braces, indicates the type of session execution.

Listing 5: The execution statement

```
1 execution{concurrent}
```

The Figure 2.3 depicts the "session side" of an interaction between multiple invoking services (clients) and a service (the server) which, while running the same code (op1), keeps session data of each client separated.

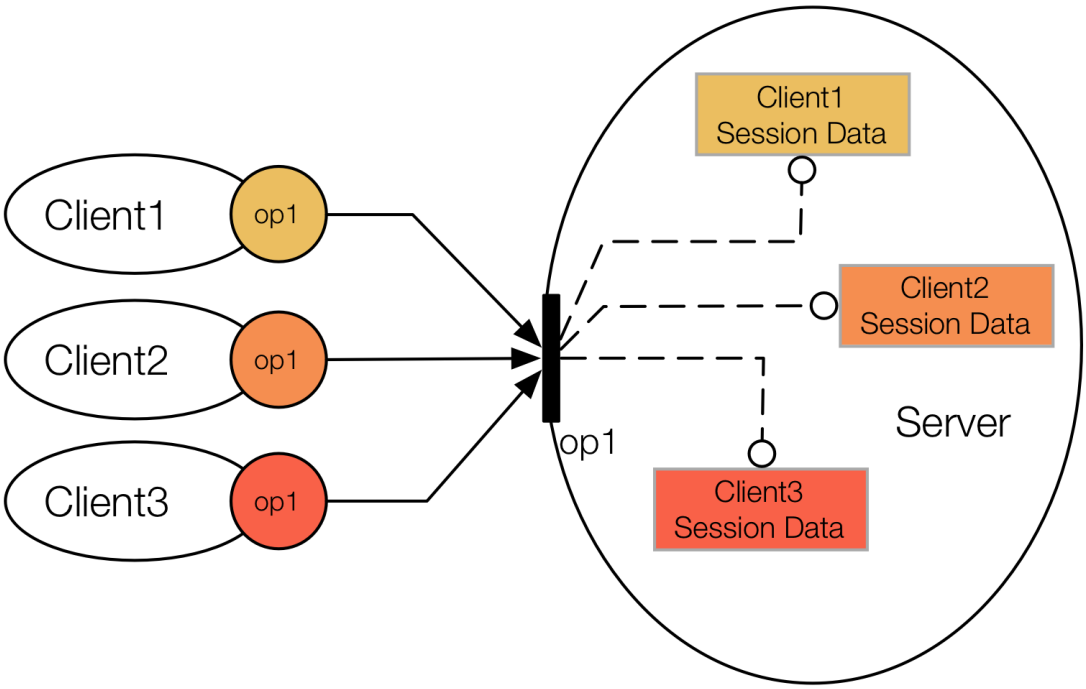


Figure 2.3: Session Data example

2.1.5.1 Correlation sets

Up to here, it's been stated that each new operation invocation, coming from another service, spawns a new session, each one with its own private data. It's noteworthy that, in a real context, invoker and invoked services rarely shares only one communication, on the contrary they usually have a "starting" communication which sets some of the parameters of the communication and then other messages can pass from a service towards the other, according to processing delay times, resources availability and the like.

This is the typical situation related to web services and sessions management: since transactions between a client, which requests some content, and the corresponding server is stateless (e.g. via HTTP protocol), to keep track of multiple activities of a certain client, HTTP cookies have been introduced. Cookies are packets of data that are sent with each client request and that carry information about session identification, user authentication and so on.

The same problem exists in services transactions and, to provide a protocol-agnostic solution, JOLIE implements natively a mechanism called *correlation sets*.

A correlation set is the set of data shared among invoker and invoked services and which is used to preserve the state of the session - or better to precisely identify it - by means of a unique concatenation of data corresponding to that session.

As stated, JOLIE implements natively the definition of correlation sets in the application preamble and this is done by means of the construct `cset`.

Thus a cset is identified by a set of nodes whose values identifies a session. Furthermore, the values contained (and passed) in the cset aren't sufficient for correlating incoming messages and sessions because the service must be informed about how to identify correlation set values within the incoming message. Thus, the cset declaration must be enhanced in a way that explicitly specifies what are the cset values that must be checked each time the service receives a message. In this way a message is correctly retained from the receiving service, only if the message nodes have the same values of the related variables.

It's noteworthy that, in order to obtain unique session identification, the services must implement a unique session identifier or a set of data whose ensemble works as a unique identifier. Figure 2.4 depicts the simple case in which the name of the clients (*c1*, *c2*, *c3*, ...) is defined as a monotonically increasing value, thus unique.

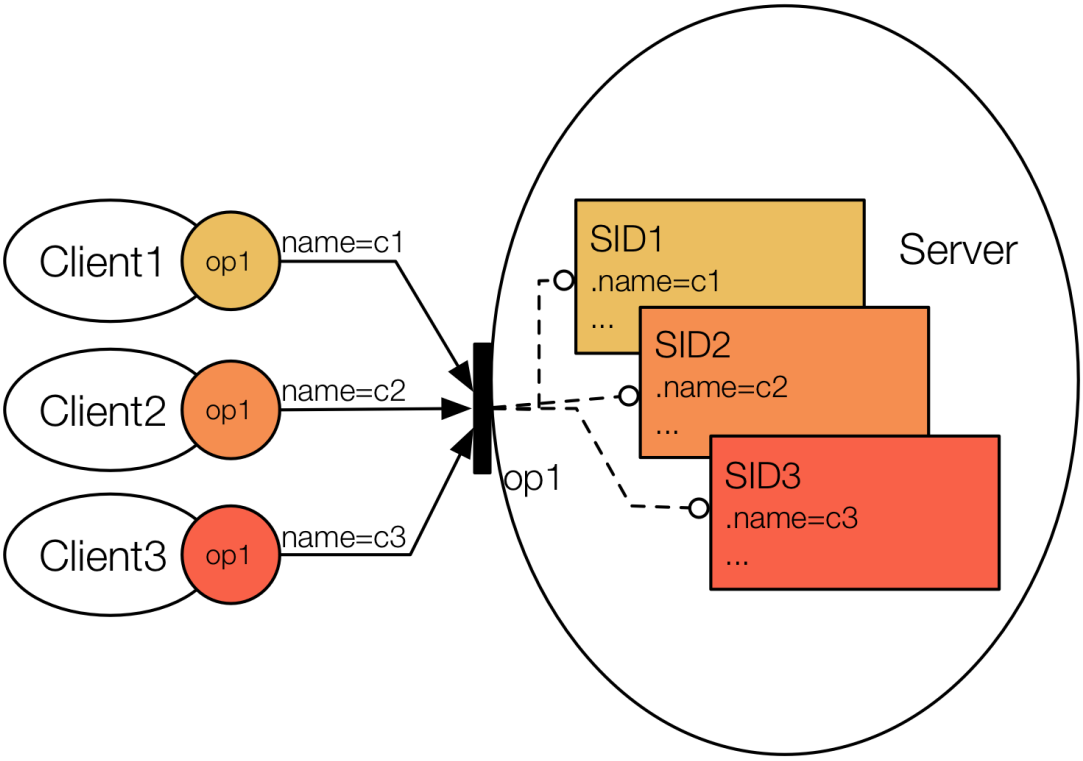


Figure 2.4: Correlation Sets example

In the example reported below a session is identified uniquely by the couple of values `name` and `surname`[3], that are located in messages incoming towards `openSession`, `makeRequest`, `closeSession` operations.

Listing 6: The cset statement

```
1 //application preamble
2 cset{
3     name:openSession.name makeRequest.name closeSession.name,
4     surname: openSession.surname makeRequest.surname closeSession.surname
5 }
```

[3]in this case it's assumed that the system has no homonyms except for names and surnames considered individually

2.1.5.2 The `init` scope

As exposed in section 2.1.4, the `main` procedure implements the service behavior (code) and handles the different values of data coming from other services. Thus, each session execute its own workflow separately. Although some applications may require to run an initialization code, which is the same for any session, and then to proceed in separate session execution. This need is met by means of the `init` scope.

This scope allows the specification of a series of statements (with the same composition rules described in section 2.1.4) which works as an initialization procedure, that is the same for each session. Thus such code is run once, when the service is launched, and never repeated.

2.1.5.3 The `global` scope

As already described, each instance of a JOLIE application is a separated session with its own private data. To share variables among multiple sessions, the language offers the definition of `global` variables. Such variables are nodes of the structure `global`, which is shared (visible and modifiable) by each session.

Listing 7: The global scope

```
1 //application preamble
2 main{
3     openSession(msg);
4     ...
5     global.db.userscount++;
6     global.db.user[global.db.userscount].name=msg.name;
7     global.db.user[global.db.userscount].surname=msg.surname;
8     ...
9 }
```

In the example reported above, the node `global.db` contains two sub-nodes, a node `userscount` used to keep track of the number of users that opened a session and the `user` array, which contains a shared structure with the names and surnames of the users, indexed by their user (arrival) count.

2.1.6 Fault Handling

Fault handlers define how an application shall respond when a certain fault (error) happens. In JOLIE `scopes`, `install` and `throw` statements are used to handle faults.

2.1.6.1 Scope, Install and Throw

Faults can be raised either by the interpreter, when encountering a fault during the execution, or by means of the `throw` statement. When a fault happens, it's caught within the scope it has been raised[4] and in such boundaries can be handled by means of the `install` statement. It's noteworthy that, if a fault isn't caught and handled within the scope it has been raised, it's automatically re-thrown to the parent scope.

As a matter of facts, the `install` statement is the command directly involved in dynamic fault handling. Such statement allows the description of a fault handling procedure (with the same composition rules described in section 2.1.4) corresponding to a specifically raised fault.

In the example shown below the `throw` statement, at the end of *myScope* `scope`, raises a *myFault* `fault` (exception), that, once caught by the corresponding `install` statement, the code associated with it (the procedure defined as *reset_settings*) is executed.

Listing 8: Fault Handling example

```
1  define reset_settings{
2      ...
3  }
4
5  main{
6      ...
7      scope(myScope){
8          install(myFault=>
9              reset_settings);
10         ...
11         throw(myFault)
12     }
13 }
```

[4] it's noteworthy that the `main` procedure itself is a scope called "main"

2.1.7 Termination

The practice of terminating faulty executions is based upon fault handling statements described in the previous section.

Termination is mechanisms intended for dealing with the recovery of a fault happened during the execution of an application. As aforementioned, the `install` statement allows the specification of a procedure which can be run only in case the corresponding fault is raised, within the scope that `install` has been declared.

In JOLIE the termination of a branch's execution is obtained by a sibling branch which can raise a fault. In such case the `install` statement takes as argument the keyword `this`, which specifies that the recovery handler is installed for the whole enclosing scope (*myScope*). It's worth noting that, if *myScope* can reach the end of its (successful) execution and the `throw` statement is run after it, no termination code is executed.

Listing 9: Termination example

```
define reset_settings{
    ...
}

main{
    ...
    scope(myScope){
        install(this=>
            reset_settings);
        ...
    }
    |
    throw(myFault)
}
```

To handle recovery of an activity that has successfully completed its execution, JOLIE offers the `comp`(*scope_name*) statement, but since it's not used in this work, its knowledge is left to the reader's curiosity.

2.2 Data structures and Flow Control operators in JOLIE

In this section are treated the main features of the language (JOLIE) about either data, data structures and flow control constructs.

2.2.1 JOLIE approach to structured data.

2.2.1.1 Basic data types and methods

JOLIE supports seven basic data types:

- 32 bit integers with sign - `int`;
- 64bit integers with sign - `long`;
- 64 bit double-precision floats - `double`;
- strings - `string`;
- empty nodes - `void`;
- any other type possible value - `any`;
- byte arrays - `raw`[5].

JOLIE is a duck-typing language, thus there's no need to declare any variable or its type in advance, since it's determined when assigned/read at runtime.

Variable assignation is made in a C/Java-like way, by means of the "=" infix operator:

[5]byte arrays can not be created directly by the programmer, but they are supported in case a service returns one and the recipient need to pass it to another service. Anyway it's not important in the context of this work, because `int`s, `double`s and `string`s are all the kind of data necessary for the purpose of patterns realization.

```
1 varA = 12;
2 varB="Hello"
```

JOLIE supports also some basic arithmetic operators like:

- *add* (+);
- *substract* (-);
- *multiply* (*) ;
- *divide* (/);
- *modulo* (%);
- *pre-increment/decrement* (++var and –var);
- *post-increment/decrement* (var++ and var–);

furthermore strings can be inputted enclosing them between double quotes, with support of the main Java escaping characters[6], space preservation (tabs, white spaces and new lines), and concatenation among several strings (or automatically cast types) by means of the + operator.

JOLIE variables' basic types can be both cast and checked by means of (respectively) int(var), double(var) and string(var) methods and is_int(var), is_double(var) and is_string(var) methods.

Besides these methods, other two constructs are added to handle variable definition checking and undefining, which are, respectively, is_defined(var) and undef(var).

[6] \' - single quote; \" - double quotes; \\ - backslash; \n - new line; \t - horizontal tab; \b - backspace.

2.2.1.2 Every JOLIE variable is a vector

It's very important to understand the approach implemented in JOLIE about data definition.

IN JOLIE EACH VARIABLE IS A VECTOR,

thus when an assignation statement is written like in the previous section:

```
varA = 12;
varB="Hello"
```

it's automatically interpreted by JOLIE like the assignation of the first (0) value of an array:

```
varA[0] = 12;
varB[0]="Hello"
```

If in first instance this approach can look confusing and complicated with relation to simple data, it becomes immediately handy when dealing with complex nested data.

Furthermore, since its strong focus on vectors, JOLIE offers a very useful vector size operator # with prefix notation:

Listing 10: Vector size operator

```
varA[0]=1;
varA[1]=2;
varA[2]=3;
varA[3]=4;
#varA // 4
```

It's worth noting that, w.r.t. the aforementioned methods, using the undefinition operator on a vector (by means of its name) erases the whole vector[7], while single vector's elements can be erased by applying the operator to them:

[7] thus undef(var) it's not interpreted like undef(var[0])

Listing 11: Undefinition operator

```
varA[0]=1;
varA[1]=2;
varA[2]=3;
varA[3]=4;
undef(varA[2]) // erases the third element of the vector
undef(varA) //erases the whole vector
```

2.2.1.3 JOLIE data structures and data structures' operators

Data structures in JOLIE have a tree-like conformation[8] in which navigation is done by using the . (dot) operator and on which the same assumptions made in the previous section still apply, thus, navigating a nested vector structure like this:

```
root.parent.son.leaf[3]
```

that means, verbosely, the fourth element in the leaf array in the structure root.parent.son, has it's automatic conversion into the statement:

```
root[0].parent[0].son[0].leaf[3]
```

verbosely, the fourth element in the leaf array, in the first element of son array, in the first element of parent array, in the first element of root array.

In order to easily handle this kind of data structures, JOLIE offers a series of operators for navigating, copying and aliasing purposes.

2.2.1.3.1 Dynamic look-up Nested variables can be identified by means of a string expression, which is evaluated at runtime. This feature is obtained by enclosing a string between round parenthesis as shown below:

[8] similar but not isomorphic to XML or JSON trees

Listing 12: Dynamic look-up operator

```
last="leaf";
root.("parent").("so"+"n").(last)[3]
```

The expression given above is interpreted in the same way described in the previous example. It's noteworthy that any kind of string concatenation aforementioned is permitted, since every dynamic look-up evaluates its content at runtime.

2.2.1.3.2 Repetitive variable paths shortcut The with operator provides a useful shortcut for repetitive variable paths:

Listing 13: With operator

```
with(animals){
    .pet[0].name="Rufus";
    .pet[0].species="cat";
    .pet[1].name="Pongo";
    .pet[1].species="dog";
    .wild[0].name="Skere Khan";
    .wild[0].species="tiger";
    .wild[1].name="Simba";
    .wild[1].species="lion"
}
```

2.2.1.3.3 Deep copy operator The deep copy operator <<[9] provides a useful shortcut for copying an entire data structure into another:

Listing 14: Deep copy operator

```
zoo.animals<<animals
```

Since the animals structure copied into zoo.animals sub-structure is the one defined in the previous section, the content of zoo.animals after the deep copy operator execution will be the same shown in 2.2.1.3.2.

[9] infix notation, the left argument is the assigned structure, the right one is the one copied.

2.2.1.3.4 Structure aliases Structure aliases are variables which point to other variables.

Aliases are created by means of the ->[10] operator. It's of fundamental importance to understand that aliases in JOLIE are evaluated every time they are used, and not only at creation (definition) time.

Thus, aliasing a variable can allow a programmer to exploit it as a mechanism for deep structure navigation (with relation to the animals structure declare in section 2.2.1.3.3):

Listing 15: Alias operator

```
1  //preamble
2  ...
3  main {
4  foreach(kind:zoo.animals){
5      species -> zoo.animals.(kind)[i].species;
6          for(i=0,i<#animals.(kind),i++){
7              println@Console(species)()
8          }
9      }
10 }
```

The code example reported above will assign the values "*pet*" and "*wild*" to the variable `kind` in the foreach operator[11], then the species variable is set as an alias which, according to i values, points to the species data of one of the animals present in the code example in section 2.2.1.3.2.

2.2.1.4 Including default interfaces

It's noteworthy that the operation println@Console(species)(), which is used to print the content of the species variable at console, is a Solicit-Response operation, which is allowed by including JOLIE's default interface "*console.iol*" in the application preamble.

[10] infix notation, the left argument is the aliasing variable, the right one is the aliased one.

[11] the foreach operator in JOLIE is used to loop through the sub-nodes of a node, since it's not used in subsequent patterns realization, but only in this example, it's not described in the language overview.

In order to exploit all the programming power of Java, JOLIE provides a mechanism to integrate Java code into a JOLIE application. This mechanism is based in writing a Java class which extends the *JavaService* class contained in the package *jolie.runtime.JavaService*, inside the JOLIE installation folder.

To help JOLIE programmers in implementing common tasks like letting the application wait for a certain time, using advanced mathematics operations and even printing the content of a variable at console (which is theoretically equal to saving its value to a file or sending it via the HTTP protocol) JOLIE comes with several interfaces like *time.iol*, *math.iol*, *console.iol*, etc. installed by default with the JOLIE interpreter.

Such interfaces can be included by simply adding the interface inclusion statement show in section 2.1.3, while their exposed operations can be called via Solicit-Responses as shown below:

Listing 16: Console, Math and Time interfaces

```
include "math.iol"
include "time.iol"
include "console.iol"

main{
    random@Math()(a);
    a=int(a*1000);
    sleep@Time(a)();
    println@Console("Slept for "+a+" ms.")()
}
```

In the example reported above, a request to the operation random at Math service is made in order to obtain a random value between 0 a 1 (extremes excluded) as the reply to the *Solicit-Response* invocation. Then the value is multiplied by 1000 and cast as integer (rounded) to be passed as the argument of the sleep operation of the Time service. The Time service will send its response (void) to the invoker after the time (in milliseconds) defined by the request is passed. Finally the println operation is sent at Console service which prints (plus a line) the concatenation of the strings “Slept for”, the number of milliseconds sent to Time service and “ms”.

2.2.2 Flow Control Operators

Execution flow in JOLIE is controlled by means of some of the classical imperative constructs.

2.2.2.1 Conditional operators

The if-then-else construct in JOLIE is the same implemented in many others languages like C and Java.

Accepted comparators are <, <=, >, >=, == and !=.

The definition of several cascading conditions is possible by means of the else if operator[12].

Listing 17: If, else if and else operators

```
...
if(first_condition){
    //CODE
}
else if(second_condition){
    //CODE
}
else{
    //CODE
}
```

2.2.2.2 Loop statements

As for conditional operators, also loop statements are the same implemented in C/Java.

[12] which is literally an else operator followed by an if

Listing 18: For and While statements

```
...
for(i=0,i<10,i++){
    //CODE REPEATED FOR TEN TIMES
};
...
j=0; flag=true;
while(flag){
    if(j<10){
        j++
    }else{
        flag=false
    }
    //CODE REPEATED FOR TEN TIMES
}
```

In the code example shown above the procedure contained in both for and while scopes is run ten times.

2.2.2.3 Synchronization statements

JOLIE provides two statements for synchronization purposes: synchronized(var) and linkIn(var)/linkOut(var).

The difference between the two statements lies in their scope of application.

linkIn and linkOut statements implements a *token-request/token-release* policy in which their argument is a variable that serves as the token. When a linkIn statement is met during a branch execution, the flow of that branch is blocked until a linkOut statement is run and the corresponding token is released. Since linkIn and linkOut variables (arguments) are normally scoped within the running session, they are used for internal synchronization.

Contrarily, the synchronized statement has been expressly implemented in JOLIE to handle races between sessions that need to access to the same shared resources.

Thus the synchronized construct accepts a variable as an argument (which serves as a token) and realizes a mutual exclusion policy which guarantees that only one process at a time can access the data subjected to the procedure.

Listing 19: Synchronization statements

```
...
synchronized(lock){
    //CODE
};
...
linkOut(lock);
linkIn(lock);
...
```

Chapter 3

Workflow Patterns for SOC

3.1 Workflow Patterns and the Workflow Patterns Initiative

The *Workflow Patterns Initiative (WPI)* is a project whose purpose is to identify the core architectural constructs in workflow technology.

Workflow technology is an approach, towards services, data and applications, focused on making connections between various resources and software applications to obtain a certain behavior. Such behavior is the result of loosely coupled applications that communicate, the one with the others, exchanging and processing data in a modular composition. Much like the one defined by SOC in the first chapter.

Each application can be seen as a construct, which realizes a particular behavior, that encapsulates algorithms and processes data, that will be used as a final output by the user or as an input for another application. This approach as been declined in the form of the *Business Process Modeling* activity, a practice of representing processes (social, economic, productive, etc.) as a composition of their interactions.

In this context the *WPI* strives to clearly define and delineate the fundamental requirements of the business process modeling by means of its recurring patterns.

The pattern based approach employed by the *WPI* has been chosen to offer both a language-and-technology-independent means of expressing each requirement's

characteristics, in the most generic form, to allow its application to the wider variety of offerings as possible.

Once identified and defined, these patterns have been described both in a imperative verbal way and through the use of a specific set of definition for the various components of the workflow system, according with the perspective in which that system is considered (i.e. control, resource, etc.).

WPI Patterns and Conventions Adopted

In the analysis of process-aware information systems, the WPI identified various perspectives:

- **Control-flow**: it captures aspects related to control-flow dependencies between various tasks (e.g. parallelism, choice, synchronization etc).
- **Data**: it deals with the passing of information , scoping of variables, and the like.
- **Resource**: it deals with resource to task allocation, delegation, etc.

Along with those describe above, another kind of perspective has been identified: the **exception handling** perspective which deals with the various causes of exception and actions that need to be put in place, as result of an occurring exception.

The WPI work on workflow patterns has culminated into an extensive collection of patterns, finely described in terms of verbal and modeling language, whose description, motivation, diagram and evaluation criteria are used in this work in order to give the strictest implementation of each pattern in JOLIE.

In this thesis work only Control-flow and Resource Pattern are taken into account because of their main importance in defining workflow paradigms.

It's also very important to underline that, all of the work done on pattern analysis and feasibility in JOLIE is based on a *keep-it-simple* principle: even if the language may allow the use of a multitude of complex but suitable advanced constructs - like aggregation, redirection and embedding -, a restricted set of "basic" constructs[1] have been employed to give the most meaningful yet straightforward demonstration of pattern feasibility in the language.

[1] described in Chapter 2

3.2 Control-Flow Patterns

3.2.1 Control-Flow Patterns and Colored Petri-Nets

Colored Petri-Nets (CPN) is a graphical language, which is used for modeling and analyzing distributed systems and, since it's focus in modeling of process-aware information systems, it's employed in workflow analysis to design communication protocols, embedded systems, distributed systems and the like.

As it follows, it's reported a brief description of the main components of CPNs, in order to provide a general knowledge of this language, to be able to interpret the representing diagram of each pattern. Finally it's explained why the CPN language is used to model Workflow Patterns and some conventions adopted in Workflow Patterns representation are defined.

3.2.1.1 From Petri-Nets to Colored Petri-Nets

Classical Petri-Nets

The *Petri-Net* (PN) is a modeling languages made for the description of distributed systems. A *PN*-net is a directed graph, in which nodes represent transitions and places. Classical PNs are made by just three elements:

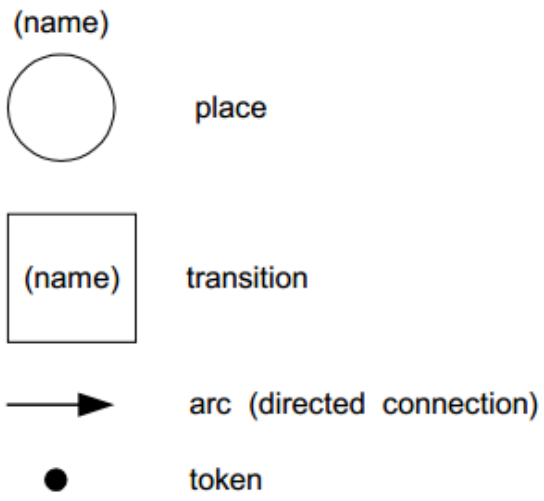


Figure 3.1: Petri-Nets Elements

- *Place:* the graphical symbol for a place is a cycle. A place represents the state of the modeled system (usually defined by the transition of a token).
- *Transition*: the graphical symbol for a transition is a box. Each transition can *fire* a modification in the state of the system. A transition is allowed to fire only if enabled, thus its preconditions must be fulfilled in order to change the position (from its input to its output) of a set of tokens whose cardinality depends on the cardinality of each incoming and outgoing arc.
- *Arc*: the graphical symbol for an arc is an arrow. An arc connects a place with a transition and vice versa. An arc coming from place to a transition means that the place condition is necessary (precondition) for the occurrence of the event. Contrariwise, an arc coming from a transition to a place means the occurrence of the place condition, thus making it true (post-condition).

As defined above, a *PN*-net is a directed graph, thus each arc (connection) is directed and there can be no connection between two places or two transition, but there's no limit about the number of arcs between two nodes.

Colored Petri-Nets

*High-Level PN*s (HLPNs) are an evolution of PNs and tackle some of their main limitations like the inability to test for zero tokens in a place, the fast-growing size of nets according to model size, the lack of temporal characterization and support for large models structuration.

One of the proposed implementations of HLPNs are the *Colored Petri-Nets* (CPNs) which support *color sets* (data type) modeling. In PNs tokens represent objects in the modeled system. Therefore, to represent attributes of these objects, PNs tokens are extend in CPNs with colored (typed) values. In CPNs the transitions determine the values of the produced colors (tokens) based on the values of the consumed colors, i.e. a transition describes the relation between the values of the input colors and the values of the output colors. Likewise preconditions are specifiable and they take the colors of tokens to be consumed into account.

Thus places in CPNs have an associated type, which determine the kind of data that the place may contain, which is usually written in *italics*, next to the place.

Colored Petri-Nets, by an example

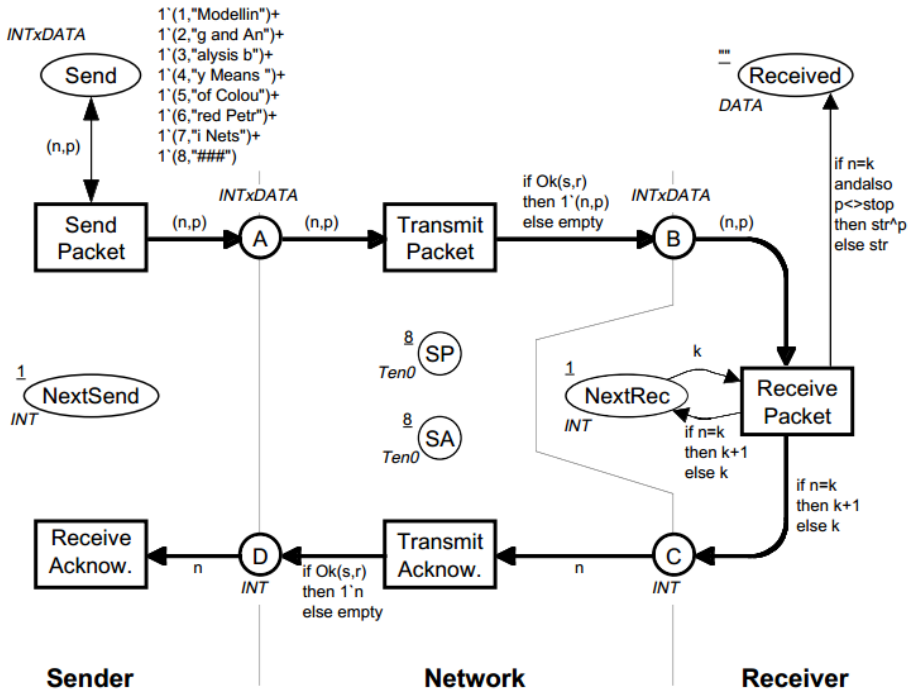


Figure 3.2: A Colored Petri-Net Example

In the example provided by Figure 3.2 places *Send*, *A*, and *B* have the type INT*x*DATA, where the *x* symbol means the cartesian product of the basic type INT(egers) and the custom type DATA. In this specific example the elements of the type represent packets to be transmitted over the Network.

Each packet is a pair, where the first element is the packet number (of type INT), while the second element is the data contents of the packet, i.e. a text string (of type DATA). During the execution each place will contain a varying number of tokens, each of which carries a data value that belongs to the type associated with the place.

The example provided by Figure 3.2 is useful also to represent another important feature of CPNs notation, that is the cardinality of tokens of the same kind, expressed with the notation `#'(token)` and defined as a *multi-set of token values*, in which the # before the ' is called the coefficient of the multi-set.

Each state represented by a CPN is defined as a marking and it consists of a number of tokens positioned on individual places, namely the marking of that place.

The convention used in this context identifies the initial marking with an underline, next to the place, albeit when the specification of the initial marking is lengthy, the underlining is omitted (like for *Send* place specification).

As in PNs, transitions are drawn as rectangles in CPNs and represent an action of removal or addition of tokens according to arcs orientation, while the number of tokens involved by the transition is determined by the arc expression (positioned next to the arc).

Still taking as an example the CPN provided by Figure 3.2, the transition *SendPacket* has three surrounding arcs, two of which share the same expression, but with different orientations. Thus they are "collapsed" and represented as a bidirectional (double) arc between *Send* place and *SendPacket* transition. By analyzing the notation that surrounds the considered arcs, it's noticeable the definition `(n,p)` which are free variables: `n` of type INT and `p` of type DATA. This to identify a particular occurrence of the transition the two free variables must be bind together such that `n` takes an INT value and `p` takes a value from DATA. If both tokens (`n` and `p`) are available the transition may occur.

In this context another assumption is made, namely the *binding element*. As defined above, to specify an occurrence of *SendPacket* from *Send*, two typed values of `n` and `p` must be chosen, bind and tested against the transition's condition. Such a couple made of a transition and the bind variables appearing on its surrounding arcs is called a binding element. W.r.t. the example of Figure 3.2, the initial marking enables the binding element (`SendPacket, <n=1, p="Modellin"`) which occurrence lead to a marking identical to the initial, except that a new token with value (`1,"Modellin"`) has been added to place *A*.

Finally a last note regarding the *TransmitPacket* transition that contains a particular conditional binding element (`n,p,s,r`) to the place *B*. This arc defines a function call `Ok(s,r)` which, according to the values of `r` and `s`, returns true if $r \leq s$.

3.2.1.2 Representing Control-Flow in Colored Petri-Nets

Solutions to problems are often non-unique, they recur in many systems, but developers invest their time on solving a problem often reinventing an already existing solutions.

A method to generalize and make a solution dependent only on general context assumptions and requirements about its behavior, possibly making known

its advantages and disadvantages, is the definition of a pattern language which can abstract an implemented solution, by means of its main features, defining a realization-agnostic pattern, which can be efficiently transposed into a model.

The language selected for representing Control-Flow Patterns by the WPI is the CPN, as it allows to model data by means of colors on the top of classical PNs, suitable for representing the behavioral logic of the control-flow.

In addition to the features of the CPN language, some assumptions are made:

- input places are labeled *i1...in*;
- output places are labeled *o1...on*;
- internal places are labeled *p1...pn*;
- transitions are labeled *A...Z*.

In case either places or transitions serve a more significant role in the context of the pattern, they are given more meaningful names (e.g. triggered-input or reset).

Unless stated otherwise, it's assumed that the tokens flowing through a CPN model that signify control-flow are typed CID (short for "Case ID") and that each executing case (i.e. process instance) has a distinct case identifier.

For most patterns, the assumption is also made that the model is safe, which means that each place in the model can only contain at most one token such that one thread of control for each case currently being executed.

3.2.2 Adopted Conventions

3.2.2.1 Dealing with simultaneous reaching branches

Control-Flow patterns deal, among with other features, with parallel and simultaneous incoming branches which, keeping an implementation-agnostic approach, can be identified as branches which reach the pattern construct at the exact same time.

Looking at the meaning of "simultaneously reaching branches", from an implementation approach, brings the definition given above into question because, dealing with time, in a discrete domain, means defining time slots and process scheduling, which declines the notion of "simultaneous" events from "processes reaching the pattern construct at the same time" to "processes reaching the pattern construct within a given time range the one w.r.t. the other".

In this work the eventuality of simultaneously reaching branches is not taken into account, since JOLIE provides mutually exclusive atomic constructs which prevent this "simultaneous reach" contingency to happen (as a matter of fact, processes that reach the pattern block at the same time are scheduled sequentially by the SO/VM thread scheduler).

A "simultaneous" approach could be taken into account once a strict definition of "simultaneous reach" is given.

3.2.2.2 Dealing with fault handling operations

Since JOLIE is mainly focused on communication among services, the most part of workflow patterns implementation is structured with a server-client approach, which makes use of OneWay and RequestResponse operations. These operations are provided with a fault exception system which yields a reliable message transmission.

For this reason, the discard, at server-side, of the corresponding operation fired by the client, raises a fault which has to be handled unless to let stopping (crashing) the client process.

The convention adopted about this matter is to implement the server part (i.e. the pattern construct) as strict as possible to the pattern definition, such that if no branch discard is allowed by the pattern definition, no fault handing shall be implemented at client-side.

Since the focus of this work is to evaluate possible workflow patterns implementation in JOLIE, fault handling is included at client-side only if strictly needed for pattern implementation purposes, otherwise a not-fault-proof version of the client (when showed) is given.

3.2.3 Basic Control-Flow Patterns

A class of workflow patterns that captures elementary aspects of process control.

3.2.3.1 Sequence

Description

An activity in a workflow process is enabled after the completion of a preceding activity in the same process.

Diagram

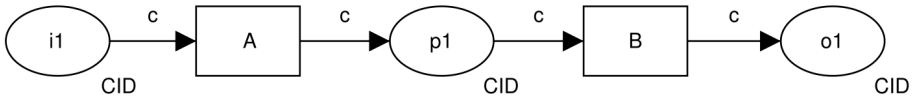


Figure 3.3: Sequence pattern

Motivation

The *Sequence* pattern is the fundamental building block for workflow processes. It's used to construct a series of consecutive activities which are executed one after the other.

Two activities take part in a *sequence* if there's a control-flow edge (statement) from one of them to the next, without guards or conditions associated with it.

JOLIE **Implementation**

JOLIE defines a specific sequential composition operator which syntax is:

```
1 Statement A ; Statement B ; ...
```

Full support for this pattern is demonstrated by any offering which is able to provide a means of specifying the execution sequence of two (or more) activities.

As said above, JOLIE offers a specific primitive operator (;) as a rule to specify the overall execution sequence, since it's basic definition no further code example is given.

3.2.3.2 Parallel Split

Description

The divergence of a branch into two or more parallel branches each of which execute concurrently.

Diagram

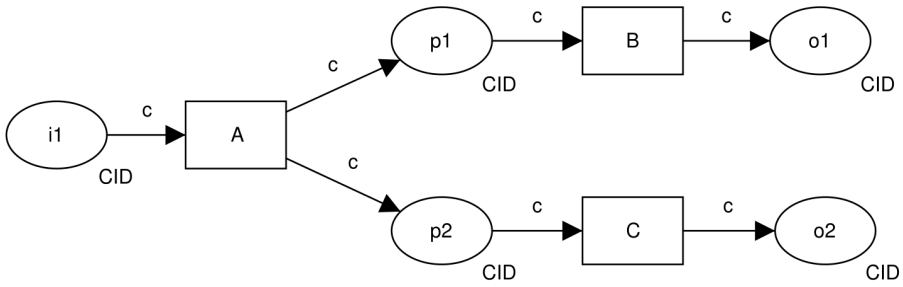


Figure 3.4: Parallel Split pattern

Motivation

The *Parallel Split* pattern allows a single thread of execution to be split into two or more branches which can execute activities concurrently.

These branches may or may not be re-synchronized at some future time.

JOLIE **Implementation**

JOLIE defines a specific parallel composition operator which syntax is:

```
1 Statement A | Statement B | ...
```

Full support for this pattern is demonstrated by any offering which is able to provide a means of specifying, at a given point of control, the thread to be split into two or more concurrent branches.

As said above, JOLIE offers a specific primitive operator (|) as a rule to specify the overall parallel execution, since it's basic definition no further code example is given.

3.2.3.3 Synchronization

Description

The convergence of two or more branches into a single subsequent branch, such that the thread of control is passed to the subsequent branch when all input branches have been enabled.

Diagram

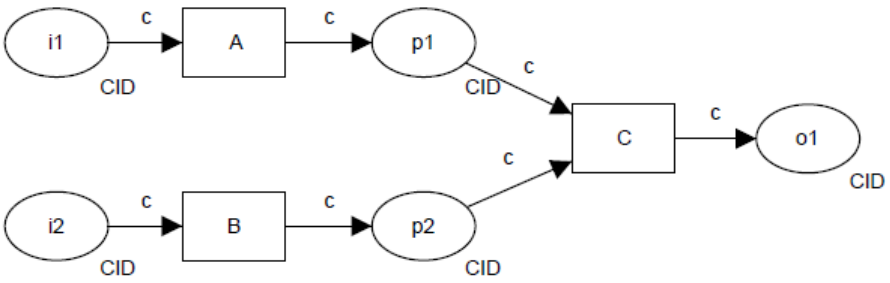


Figure 3.5: Synchronization pattern

Motivation

Synchronization provides a means of re-converging the execution threads of two or more parallel branches.

In general, these branches are created from a *Parallel Split*[3.2.3.2] construct earlier in the process model. The thread of control is passed to the activity immediately following the synchronizer, once all of the incoming branches have completed.

JOLIE Implementation

The *Synchronization* pattern raises two important context conditions associated with it:

1. each incoming branch executes precisely once for a given case;
2. the synchronizer can only be reset (and fire again) once each incoming branch has completed.

These conditions are important since if all incoming branches do not complete, then the synchronizer will deadlock and if more than one trigger is received on a branch, then the behavior of the construct is undefined. They also serve to alleviate the concern as to whether all of the threads being synchronized relate to the same process instance. This issue becomes a significant problem in joins that do not have these restrictions.

JOLIE defines two specific synchronization complementary statements which syntax is:

```
linkOut(id)
linkIn(id)
```

The `linkIn` statement is the language blocking primitive, which waits for the corresponding `linkOut` operation to fire a release operation on its same *id*, while doing this, it keeps in a blocked state the process branch it takes part in.

Full support for this pattern is demonstrated by any offering which provides a construct to merge several distinct threads of execution in different branches into a single thread of execution in a single branch.

The merge occurs when a thread of control has been received on each of the incoming branches.

JOLIE **code example**

Listing 20: Synchronization code example

```
include "console.iol"
include "time.iol"

main{
    scope(pid1){
        sleep@Time(3000)();
        linkOut(idP1)
    }
    |
    scope(pid2){
        sleep@Time(4000)();
        linkOut(idP2)
    }
    |
    scope(sync){
        {
        linkIn(idP1);
```

```
18          println@Console("pid1 linked out")()
19          }|{
20          linkIn(idP2);
21          println@Console("pid2 linked out")()}
22      };
23      println@Console(
24          "pid1 and pid2 succesfully synchornized")()
25  }
```

It's worth noting that the pattern realization given above can be even simpler because of scope's characterization in JOLIE. In the language the flow control within a scope is released (and the scope exited) only if each of its branch has completed. Thus an alternative solution could involve the use of a scope (e.g. `p_container`) containing the `pid1` and `pid2` scopes composed in parallel. When both `pid1` and `pid2` executions have completed the scope `p_container` results as completed and the subsequent operation, after the synchronization of all the branches, can run.

3.2.3.4 Exclusive Choice

Description

The divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on the outcome of a logical expression associated with the branch.

Diagram

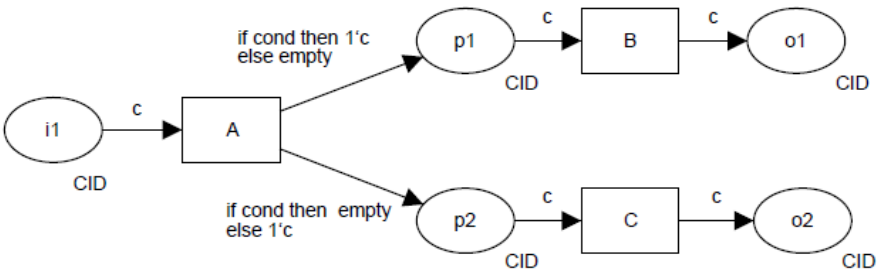


Figure 3.6: Exclusive Choice pattern

Motivation

The *Exclusive Choice* pattern allows the thread of control to be directed to a specific activity depending on the outcome of a preceding activity, the values of elements of specific data elements in the workflow or the results of a user decision.

The routing decision is made dynamically allowing it to be deferred to the latest possible moment at runtime.

JOLIE **Implementation**

The *Exclusive Choice* pattern rises two context conditions:

1. the information required to calculate the logical conditions on each of the outgoing branches must be available at runtime at the point at which the choice construct is reached in the process;

2. the condition associated with precisely one outgoing branch of the exclusive choice construct must evaluate to true.

JOLIE defines both some classical imperative flow control constructs like `if`, `else if` and `else` operators and input-guarded non-deterministic choices.

The former are mainly involved into the flow control of a specific process, the latter is used in process composition, by which if one of the input (guarded) statements in the choice receives a message, the specified process is executed, while all other possible branches are discarded:

```
if(cond1){
    process1}
else if(cond2){
    process2}
...
else{
    processN
}
```

```
[oneWayOp1(msg)]{
    process1}
[oneWayOp2(msg)]{
    process2}
...
[requestResponseN(msg)(rsp)]{
    processN}
```

Full support for this pattern is evidenced by an offering which provides a construct which enables the thread of control to be directed to exactly one of several outgoing branches. The decision as to which branch is selected is made at runtime on the basis of specific conditions associated with each of the branches.

Since two possible interpretation and solution have been suggested for this pattern, they are both listed.

JOLIE **code example**

Listing 21: Exclusive Choice code example (if-then-else)

```
include "console.iol"

inputPort EC_ite{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: rcv_msg
}

main{
    rcv_msg(msg_v);
    if(msg_v<=13){
        println@Console("Good Morning")()
    }
    else if(msg_v>13 && msg_v<=18){
        println@Console("Good Afternood")()
    }
    else if(msg_v>18 && msg_v<20){
        println@Console("Good Evening")()
    }
    else{
        println@Console("Good Night")()
    }
}
```

Listing 22: Exclusive Choice code example (non-deterministic choice)

```
include "console.iol"

inputPort EC_ite{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: morning_msg, evening_msg,
        afternoon_msg, night_msg
}

main{
    [morning_msg()]{
        println@Console("Good Morning")()
    }
    [afternoon_msg()]{
        println@Console("Good Afternood")()
    }
    [evening_msg()]{
        println@Console("Good Evening")()
    }
    [night_msg()]{
```

```
21          println@Console("Good Night")()
22      }
23  }
```

3.2.3.5 Simple Merge

Description

The convergence of two or more branches into a single subsequent branch.

Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

Diagram

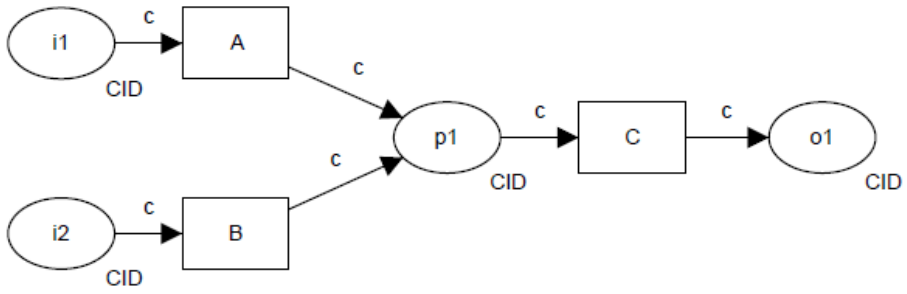


Figure 3.7: Simple Merge pattern

Motivation

The *Simple Merge* pattern provides a means of merging two or more distinct branches without synchronizing them. As such, this presents the opportunity to simplify a process model by removing the need to explicitly replicate a sequence of activities that is common to two or more branches. Instead, these branches can be joined with a simple merge construct and the common set of activities need only to be depicted once in the process model.

JOLIE **Implementation**

The *Simple Merge* pattern means no consideration of synchronization and the place at which the merge occurs is safe and can never contain more than one token.

Within JOLIE can be easily defined a looping OneWay operation that constitutes a shared but unique and safe place for incoming messages from other processes.

When a message is received the message data is processed, after which the system is ready for another message to come.

Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption.

In this case an accounting service is used as example: the looping OneWay operation `acc_calc` waits for an "income" message, after the reception the data is elaborated and stored, after which the system returns into a wait state for the next message.

An alternative approach achieving the same result can be obtained using the `execution {concurrent}` statement and a globally shared variable for `net_of_tax` and `tot_tax` accumulation.

JOLIE **code example**

Listing 23: Simple Merge code example

```
include "console.iol"

inputPort SM{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: acc_calc
}

main{
    net_of_tax=0.0;
    tot_tax=0.0;
    while(true){
        acc_calc(inc_rep);
        income=double(inc_rep.income);
        tax_perc=double(inc_rep.tax_rate);
        println@Console("New income: "+income)();
        tot_tax=tot_tax+(income*tax_perc);
        net_of_tax=net_of_tax+income*(1.0-tax_perc);
        println@Console(
            "Net of taxes: "+net_of_tax+
            ", total taxes: "+tot_tax)()
    }
}
```

3.2.4 Advanced Branching and Synchronization Patterns

The *Advanced Branching and Synchronization Patters* constitute a series of patterns which characterize more complex branching and merging concepts which arise in business processes.

Although relatively commonplace in practice, these patterns are often not directly supported or even able to be represented in many commercial offerings.

3.2.4.1 Multi-Choice

Description

The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches.

Diagram

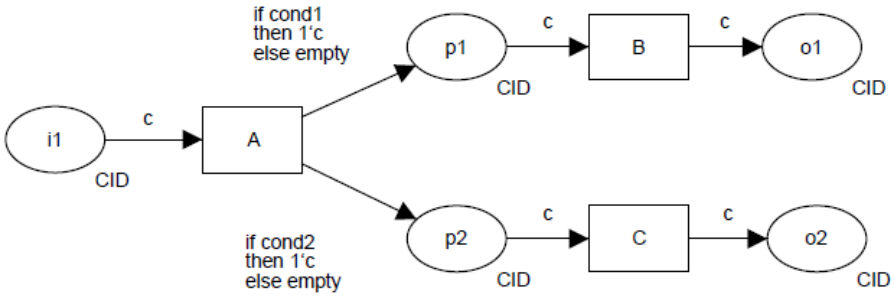


Figure 3.8: Multi-Choice pattern

Motivation

The *Multi-Choice* pattern provides the ability for the thread of execution to be diverged into several concurrent threads on a selective basis. The decision as to whether to pass the thread of execution to a specific branch is made at runtime.

It can be based on a variety of factors including the outcome of a preceding task, the values of elements of specific data elements in the process, the results of evaluating an expression associated with the outgoing branch or some other form of programmatic selection mechanism. This pattern is essentially an analogue of the *Exclusive Choice* [3.2.3.4] pattern in which multiple outgoing branches can be enabled.

JOLIE **Implementation**

The mechanism that evaluates the *Multi-Choice* is able to access any required data elements or necessary resources when determining which of the outgoing branches the thread of control should be routed.

As seen for the Exclusive Choice pattern, JOLIE allows the implementation of such a behavior both internally and externally, by means of if-then-else operators and non-deterministic choice statements. It's worth noting that, even if the *Multi-Choice* pattern condition states the availability of any required data elements or resource, even the external solution that employs the non-deterministic choice allows such availability through JOLIE data structures.

Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption.

A work-around that can be used to support the pattern in most offerings is based on the use of an AND-split immediately followed by an (binary) XOR-split in each subsequent branch.

Another is the use of an XOR-split with an outgoing branch for each possible task combination, e.g. a Multi-Choice construct with outgoing branches to tasks A and B would be modeled using an XOR-split with three outgoing branches - one to task A, another to task B and a third to an AND-split which then triggered both tasks A and B.

Note that the work-around based on XOR-splits and AND-splits is not considered to constitute support for this pattern as the decision process associated with evaluation of the Multi-Choice is divided across multiple split constructs.

One of the possible approaches to the solution to this pattern in JOLIE is employing the if-then-else structure along with parallel operator. The behavior obtained with this implementation is similar to conditions on the arcs or parallel conditions on outgoing transitions used by BPMN and BPEL language.

The following code example represents a *Multi-Choice* approach for a `cart` discount calculation based on parallel choices over the same data structure. The calculation is made on what kind of items are inside the cart (note that if the cart is empty the exit branch is 0).

JOLIE **code example**

Listing 24: Multi-Choice code example

```
include "console.iol"
include "time.iol"

init{
    discount_perc=double(0);
    with(cart){
        .drinks[0]="coke";
        .drinks[1]="water";
        .drinks[2]="tea";
        .food[0]="hamburger";
        .food[1]="hot-dog"
    }
}

main{
    if (#cart.drinks>0){
        discount_perc=discount_perc+0.02
    }|
    if (#cart.food>0){
        discount_perc=discount_perc+0.01
    }|
    if (#cart.furnishing>0){
        discount_perc=discount_perc+0.05
    }|
    {println@Console(
        "Your discount is: "+discount_perc+"%")();
     sleep@Time(2000)();
    println@Console("Your discount is: "+discount_perc+"%")()
    }
}
```

It's worth noting that in the example above, no synchronization technique is used to prevent possible errors due to read-write access to the `discount_perc` by concurrent processes. To show this behavior two printing statement, executed in parallel along with *Multi-Choice* block, have been included in the code example, in most part of executions the output of this example is an immediate "0%" value, followed by a "0.03%" value printed after 2 seconds (circa 2000ms).

3.2.4.2 Structured Synchronizing Merge

Description

The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled.

The *Structured Synchronizing Merge* occurs in a structured context, i.e. there must be a single *Multi-Choice* [3.2.4.1] construct earlier in the process model with which the *Structured Synchronizing Merge* is associated with and it must merge all of the branches emanating from the *Multi-Choice*. These branches must either flow from the *Structured Synchronizing Merge* without any splits or joins or they must be structured in form (i.e. balanced splits and joins).

Diagram

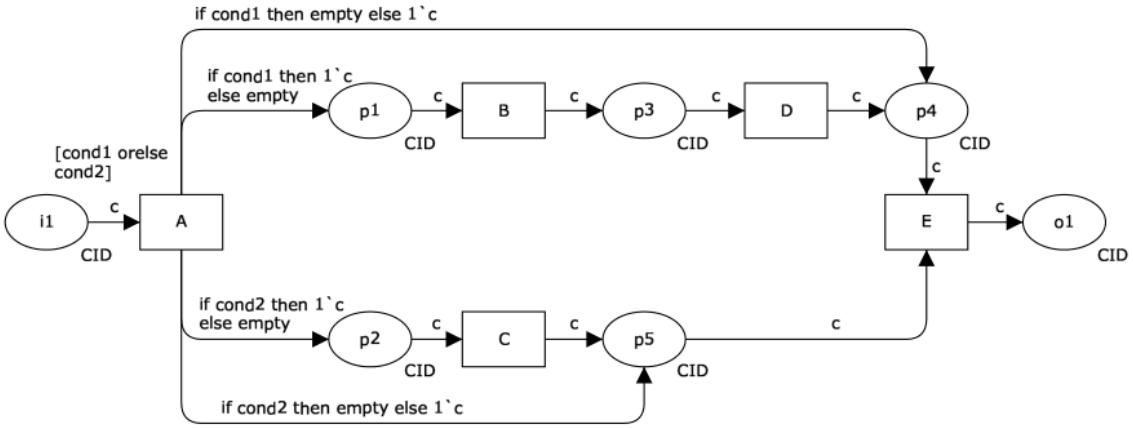


Figure 3.9: Structured Synchronizing Merge pattern

Motivation

The *Synchronizing Merge* pattern provides a means of merging the branches resulting from a specific *Multi-Choice* or *Exclusive Choice* construct earlier in a workflow

process into a single branch. Implicit in this merging is the synchronization of all of the threads of execution resulting from the preceding *Multi-Choice*.

JOLIE **Implementation**

As already indicated, the *Synchronizing Merge* construct provides a means of merging the branches from a preceding *Multi-Choice* construct and synchronizing the threads of control flowing along each of them.

It is not necessary that all of the incoming branches to the *Synchronizing Merge* are active in order for the construct to be enabled, however all of the threads of control associated with the incoming branches must have reached the *Synchronizing Merge* before it can fire.

As such there are four context conditions associated with the use of this pattern:

1. There must be a single *Multi-Choice* construct earlier in the process model with which the *Synchronizing Merge* is associated with and it must merge all of the branches emanating from the Multi-Choice.
   These branches must either flow from the *Multi-Choice* to the *Synchronizing Merge* without any splits or joins or they must be structured in form (i.e. balanced splits and joins) such that it is not possible for the *Synchronizing Merge* to receive multiple triggers on the same branch once the *Multi-Choice* has been enabled;

2. The *Multi-Choice* construct must not be re-enabled before the associated *Synchronizing Merge* construct has fired;

3. Once the *Multi-Choice* has been enabled none of the activities in the branches leading to the *Synchronizing Merge* can be canceled before the merge has been triggered.
   The only exception to this is that it is possible for all of the activities leading up to the *Synchronizing Merge* to be canceled;

4. The *Synchronizing Merge* must be able to resolve the decision as to when it should fire, based on local information available to it during the course of execution.
   Critical to this decision is knowledge of how many branches emanating from the *Multi-Choice* are active and require synchronization.

Addressing the last of the context conditions without introducing non-local semantics for the *Synchronizing Merge* can be achieved in several ways:

1. including the process model following a *Multi-Choice* such that the subsequent *Synchronizing Merge* will always receive precisely one trigger on each of its incoming branches and no additional knowledge is required to make the decision as to when it should be enabled;
2. by providing the merge construct with knowledge of how many incoming branches require synchronization;
3. by undertaking a thorough analysis of possible future execution states to determine when the *Synchronizing Merge* can fire.

The first of these implementation alternatives forms the basis for this pattern and it involves adding an alternate "bypass" path around each branch from the *Multi-Merge* to the *Synchronizing Merge* which is enabled in the event that the normal path is not chosen.

The "bypass" path is merged with the normal path for each branch prior to the *Synchronizing Merge* construct ensuring that it always gets a trigger on all incoming branches and can hence be implemented as an *Synchronization* construct.

Made on top of the *Multi-Choice* JOLIE implementation proposed above, the solution of the *Synchronizing Merge* pattern uses the *Multi-Choice* structure, where each branch is evaluated concurrently (AND-split); after the evaluation each of them fires a `linkOut` event that's caught from the *Synchronizing Merge* process, which is executed in parallel along with the *Multi-Choice* process.

Full support for this pattern in an offering is evidenced by the availability of a construct which demonstrates all of the context requirements for this pattern. Any offering which allows the threads of control in any subset of the input branches to the merge to be canceled before it is triggered achieves a rating of partial support.

As defined above, one of the possible approaches to the solution to this pattern in JOLIE is employing the *Multi-Choice* structure used as a solution for *Multi-Choice* pattern example, along with parallel operator and the language's synchronizing functions [3.2.3.3].

The following code example evolves the *Multi-Choice* example where a `cart` lists each kind of item purchased by a customer, its quantity and cost. Based on these

information, partial values of total amount to be paid and total applicable discount are calculated parallelly, while the final value of total amount to be paid is calculated only when all of the fired choices reached the *Synchronization Merge* structure after the corresponding `linkOut` statement.

JOLIE **code example**

Listing 25: Structured Synchronizing Merge code example

```
include "console.iol"
include "time.iol"

init{
    discount_perc=0.0;
    tot_to_pay=0.0;
    with(cart){
            .drinks[0].name="coke";
            .drinks[0].price=2;
            .drinks[0].quantity=6;

            .drinks[1].name="water";
            .drinks[1].price=1;
            .drinks[1].quantity=6;

            .drinks[2].name="tea";
            .drinks[2].price=3;
            .drinks[2].quantity=3;

            .food[0].name="hamburger";
            .food[0].price=5;
            .food[0].quantity=1;

            .food[1].name="hot-dog";
            .food[1].price=2;
            .food[1].quantity=4
    }
}

main{
    {if (#cart.drinks>0){
        drink->cart.drinks[i];
        for(i=0,i<#cart.drinks,i++){
            synchronized(lock){
                tot_to_pay=tot_to_pay+
                drink.quantity*
                drink.price
        }};
        discount_perc=discount_perc+0.02
    };
    linkOut(drinks)
    }|{if (#cart.food>0){
        food->cart.food[j];
```

```
44          for(j=0,j<#cart.food,j++){
45              synchronized(lock){
46                  tot_to_pay=tot_to_pay+
47                  food.quantity*
48                  food.price
49          }};
50          discount_perc=discount_perc+0.01
51      };
52      linkOut(food)
53      }|{if (#cart.furnitures>0){
54          furniture->cart.furnitures[y];
55          for(y=0,y<#cart.furnitures,y++){
56              synchronized(lock){
57                  tot_to_pay=tot_to_pay+
58                  furniture.quantity*
59                  furniture.price
60          }};
61          discount_perc=discount_perc+0.04
62      };
63      linkOut(furnitures)
64      }|{
65      linkIn(drinks)|linkIn(food)|linkIn(furnitures)};
66      println@Console("Total to pay $: "+
67          (tot_to_pay*(1.0-discount_perc))+
68          "\nDiscount %: "+discount_perc+
69          "\nDiscount $: "+tot_to_pay*discount_perc)()
70  }
```

3.2.4.3 Multi-Merge

Description

The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

Diagram

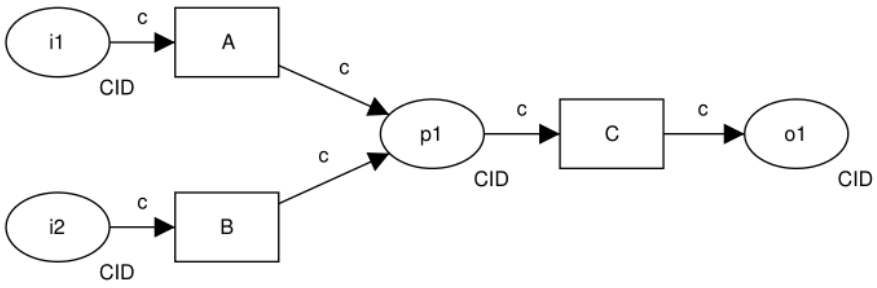


Figure 3.10: Multi-Merge pattern

Motivation

The *Multi-Merge* pattern provides a means of merging distinct branches in a process into a single branch. Although several execution paths are merged, there is no synchronization of control-flow and each thread of control which is currently active in any of the preceding branches will flow unimpeded into the merged branch.

The distinction between this pattern and the *Simple Merge*[3.2.3.5] is that it is possible for more than one incoming branch to be active simultaneously and there is no necessity for merging place to be safe.

JOLIE **Implementation**

An offering achieves full support if it satisfies the context criterion for the pattern. Partial support is awarded to offerings that do not provide support for multiple

branches to merge simultaneously or do not provide for preservation of all threads of control where this does occur.

Since its resemblance to the *Simple Merge* patter, the same example is used to underline the interesting difference between the two.

While the *Simple Merge* pattern implementation has been based on a single looping instance to define a safe merging place (the process itself) which can serve only one process at a time, the implementation of the not-safe-place *Multi-Merge* pattern can exploit the concurrent execution of multiple instances of the same process offered by JOLIE.

Synchronization and global variable state are used along with `init{}` and the concurrent execution, thus providing safe concurrent variable access and shared scope among sessions. The `spawn` construct is used at client side to run several parallel branches towards the *Multi-Merge* block.

JOLIE code example

Listing 26: Multi-Merge (server) code example

```
include "console.iol"
inputPort SM{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: acc_calc
}

execution{concurrent}

init{
    net_of_tax->global.income.net_of_tax;
    tot_tax->global.income.tot_tax;
    net_of_tax=0.0;
    tot_tax=0.0
}

main{
    acc_calc(inc_rep);
    income=double(inc_rep.income);
    tax_rate=double(inc_rep.tax_rate);
    println@Console("New income: "+income+
        " taxed at: "+tax_rate*100+"%")();
    synchronized(lock){
        tot_tax=tot_tax+income*tax_rate;
        net_of_tax=net_of_tax+
            income*(1-tax_rate)
    };
    println@Console(
```

```
29          "Net of taxes: "+net_of_tax+
30          ", total taxes: "+tot_tax)()
31  }
```

Listing 27: Multi-Merge (client) code example

```
1   include "console.iol"
2
3   outputPort SM{
4       Location: "socket://localhost:8000"
5       Protocol: sodep
6       OneWay: acc_calc
7   }
8
9   main{
10      registerForInput@Console()();
11      inc_rep.income=200;
12      inc_rep.tax_rate=0.2;
13      spawn(i over 10) in arr{
14          SM << arr[i];
15          acc_calc@SM(inc_rep)
16      }
17  }
```

3.2.4.4 Structured Discriminator

Description

The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model.

The thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on.

The discriminator construct resets when all incoming branches have been enabled.

Diagram

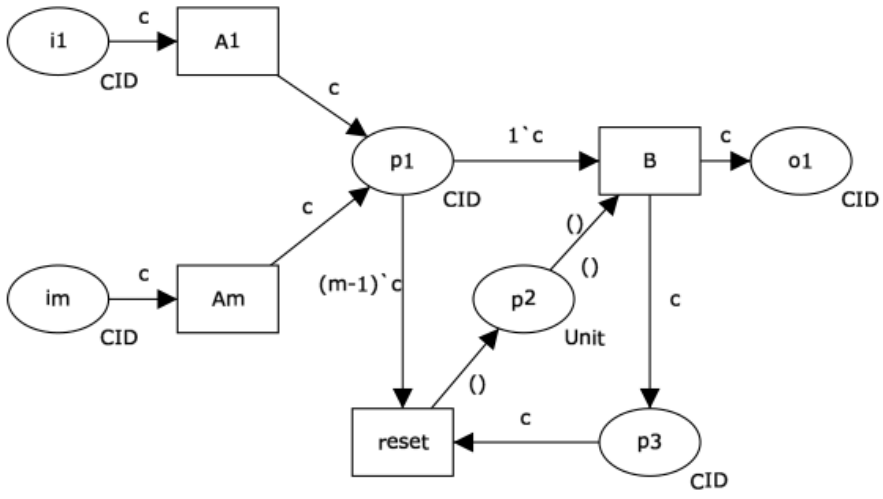


Figure 3.11: Structured Discriminator pattern

Motivation

The *Discriminator* pattern provides a means of merging two or more distinct branches in a process into a single subsequent branch such that the first of them to complete results in the subsequent branch being triggered, but completions of other incoming branches thereafter have no effect on (and do not trigger) the subsequent branch. As such, the *Discriminator* provides a mechanism for progressing

the execution of a process once the first of a series of concurrent activities has completed.

JOLIE **Implementation**

The *Discriminator* pattern provides a means of merging two or more branches in a workflow and progressing execution of the workflow as rapidly as possible by enabling the subsequent (merged) branch as soon as a thread of control is received on one of the incoming branches.

There are five context conditions associated with the use of this pattern:

1. The *Discriminator* is associated with precisely one *Parallel Split*[3.2.3.2] earlier in the process and each of the outputs from the *Parallel Split* is an input to the *Discriminator*;

2. The branches from the *Parallel Split* to the *Discriminator* are structured in form and any splits and merge in the branches are balanced;

3. Each of the incoming branches to the *Discriminator* must only be triggered once prior to it being reset;

4. The *Discriminator* resets (and can be re-enabled) once all of its incoming branches have been enabled precisely once;

5. Once the *Parallel Split* has been enabled none of the activities in the branches leading to the *Discriminator* can be canceled before the join has been triggered. The only exception to this is that it is possible for all of the activities leading up to the *Discriminator* to be canceled.

The *Structured Discriminator* can be directly implemented by specifying a custom trigger condition for an activity with multiple incoming routers which only fires when the first router is enabled.

An offering achieves full support if it satisfies the context criteria for the pattern. It rates as partial support if the *Discriminator* can reset without all activities in incoming branches having run to completion.

The JOLIE code example written following is structured into two processes (files), server and client, which provide both the *Synchronization* pattern (client) and a concurrent implementation of the *Structured Discriminator* pattern (server).

Multiple executions of the same "server" process are allowed by the concurrent execution statement (correlation sets are implemented too due to JOLIE session management requirements).

To simulate different and non-deterministic branches execution times, each branch waits a random number of seconds before sending a OneWay request towards the server, which implements reset-after-synchronization policy on top of the JOLIE code example given for the *Synchronization* pattern[3.2.3.3].

JOLIE **code example**

Listing 28: Structured Discriminator (server) code example

```
include "console.iol"
include "math.iol"
include "time.iol"

execution{concurrent}

type session: undefined
type a1: undefined
type a2: undefined
type a3: undefined

cset{
    id: session.id
        a1.id
        a2.id
        a3.id
}


inputPort SD{
    Location:"socket://localhost:8000"
    Protocol: sodep
    OneWay: alert1(a1), alert2(a2), alert3(a3), start(session)
}

define printFirst {
    println@Console(
        "First alert received "+
        "from the "+token+" circuit")()
}

main{
    [start(session)]{
        println@Console("--- Serving session "+
            session.id+" ---")();
        {scope(alert_sensing){
            alert1(a1);
            if(!is_defined(token)){
                    token="First";
                    token.time=a1.time;
```

```
                    printFirst
                };
                println@Console("One: "+a1.time)();
                linkOut(alert1)
            }|{
                alert2(a2);
                if(!is_defined(token)){
                    token="Second";
                    token.time=a2.time;
                    printFirst
                };
                println@Console("Two: "+a2.time)();
                linkOut(alert2)
                }|{
                    alert3(a3);
                    if(!is_defined(token)){
                        token="Third";
                        token.time=a3.time;
                        printFirst
                };
                println@Console("Three: "+a3.time)();
                linkOut(alert3)
                }
            }|{
                linkIn(alert1)|linkIn(alert2)|linkIn(alert3)
            };
            println@Console("All alert received")()
        }
}
```

Listing 29: Structured Discriminator (client) code example

```
include "console.iol"
include "math.iol"
include "time.iol"

outputPort SD{
    Location:"socket://localhost:8000"
    Protocol: sodep
    OneWay: alert1, alert2, alert3, start
}

main{
    while(true){
        scope(ini){
            range=10;
            getCurrentDateTime@Time()(start_time);
            random@Math()(a1.time);
            random@Math()(a2.time);
            random@Math()(a3.time);
```

```
19              random@Math()(session.id);
20
21              a1.timems=int(a1.time*range)*1000;
22              a2.timems=int(a2.time*range)*1000;
23              a3.timems=int(a3.time*range)*1000;
24
25              session.id=int(session.id*10000);
26              a1.id=session.id;
27              a2.id=session.id;
28              a3.id=session.id;
29
30              a1.time=int(a1.timems/1000);
31              a2.time=int(a2.timems/1000);
32              a3.time=int(a3.timems/1000);
33              start@SD(session);
34              println@Console("\nStarting session: "+session.id)()
35          };
36          scope(exec){
37              {
38                  sleep@Time(a1.timems)();
39                  println@Console("sending alarm 1")();
40                  alert1@SD(a1)
41              }
42              |
43              {
44                  sleep@Time(a2.timems)();
45                  println@Console("sending alarm 2")();
46                  alert2@SD(a2)
47              }
48              |
49              {
50                  sleep@Time(a3.timems)();
51                  println@Console("sending alarm 3")();
52                  alert3@SD(a3)
53              }
54          }
55      }
56  }
```

3.2.4.5 Blocking Discriminator

Description

The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model.

The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. The *Blocking Discriminator* construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the *Blocking Discriminator* has reset.

Diagram

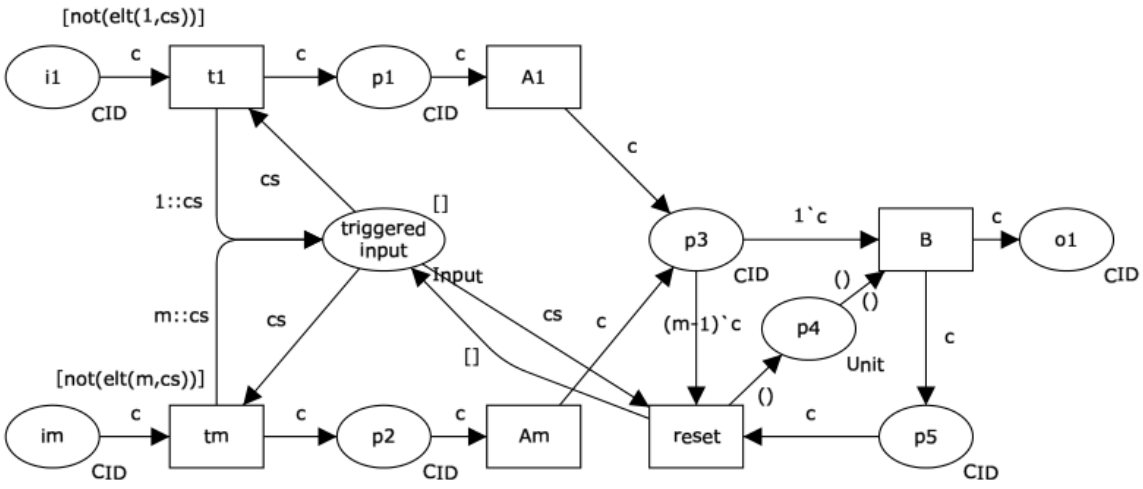


Figure 3.12: Blocking Discriminator pattern

Motivation

The *Blocking Discriminator* pattern is a variant of the *Structured Discriminator* [3.2.4.4] pattern that is able to run in environments where there are potentially several concurrent execution threads within the same process instance. This quality allows

it to be used in loops and other process structures where more than one execution thread may be received in a given branch in the time between the first branch being enabled and the *Blocking Discriminator* being reset.

JOLIE Implementation

The *Blocking Discriminator* pattern is more robust than the *Structured* one, as it is not subject to the constraint that each incoming branch can only being triggered once prior to reset.

The *Blocking Discriminator* functions by keeping track of which inputs have been triggered (via the triggered input place) and preventing them from being re-enabled until the construct has reset as a consequence of receiving a trigger on each incoming branch.

An important feature of this pattern is that it is able to be used in environments that do not support a safe process model or those that may receive multiple triggerings on the same input place e.g. where the *Blocking Discriminator* is used within a loop.

The JOLIE code example for this pattern has been written as an slight evolution of the one used for the *Structured Discriminator* pattern. In this case the limitation of simultaneous *Discriminator* executions during subsequent processing is obtained by using the `execution{sequential}` statement which starts a new session only when the preceding one is finished.

JOLIE code example

Listing 30: Blocking Discriminator (server) code example

```
include "console.iol"
include "math.iol"
include "time.iol"

execution{sequential}

type session: undefined
type a1: undefined
type a2: undefined
type a3: undefined

cset{
    id: session.id
        a1.id
        a2.id
        a3.id
}

```

```
19
20 inputPort SD{
21     Location:"socket://localhost:8000"
22     Protocol: sodep
23     OneWay: alert1(a1), alert2(a2), alert3(a3), start(session)
24 }
25
26 define printFirst {
27     println@Console(
28         "First alert received "+
29         "from the "+token+" circuit")()
30 }
31
32 main{
33     [start(session)]{
34         println@Console("--- Serving session "+
35             session.id+" ---")();
36         {scope(alert_sensing){
37             alert1(a1);
38             if(!is_defined(token)){
39                     token="First";
40                     token.time=a1.time;
41                     printFirst
42             };
43             println@Console("One: "+a1.time)();
44             linkOut(alert1)
45         }|{
46             alert2(a2);
47             if(!is_defined(token)){
48                     token="Second";
49                     token.time=a2.time;
50                     printFirst
51             };
52             println@Console("Two: "+a2.time)();
53             linkOut(alert2)
54             }|{
55                 alert3(a3);
56                 if(!is_defined(token)){
57                     token="Third";
58                     token.time=a3.time;
59                     printFirst
60             };
61             println@Console("Three: "+a3.time)();
62             linkOut(alert3)
63             }
64         }|{
65             linkIn(alert1)|linkIn(alert2)|linkIn(alert3)
66         };
67         println@Console("All alert received")()
68     }
69 }
```

3.2.4.6 Canceling Discriminator

Description

The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model.

The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. Triggering the *Cancelling Discriminator* also cancels the execution of all of the other incoming branches and resets the construct.

Diagram

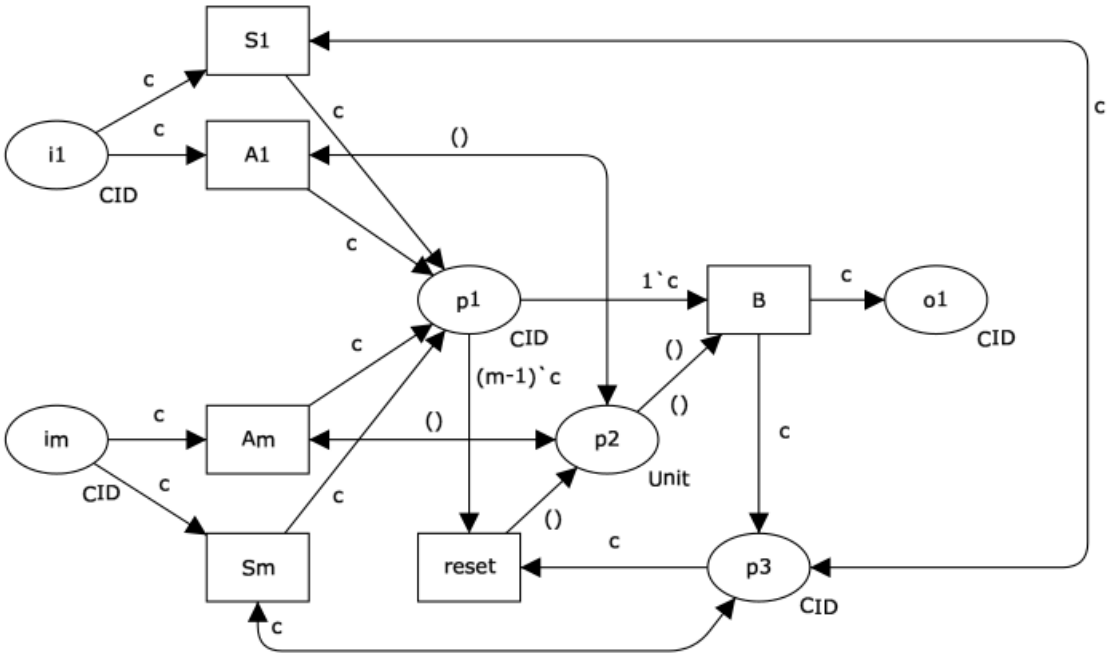


Figure 3.13: Canceling Discriminator pattern

Motivation

This pattern provides a means of expediting a process instance where a series of incoming branches to a join need to be synchronized but it is not important that the tasks associated with each of the branches (other than the first of them) be completed.

JOLIE Implementation

Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. An offering is considered to provide partial support for the pattern if there are side-effects associated with the execution of the pattern (e.g. tasks in incoming branches which have not completed being recorded as complete).

The JOLIE code example for this pattern has been written as an evolution of the one used for the *Structured Discriminator* pattern, but in this case a non-deterministic choice approach is used. When a non-deterministic case reach the *Canceling Discriminator* the other branches are still waiting to be accepted, that eventuality should be handled at client-side as stated in adopted conventions [3.2.2].

The server part of the *Cancelling Discriminator* pattern is listed as follows, while for the client part is made reference to the *Standard Discriminator* implementation.

JOLIE code example

Listing 31: Canceling Discriminator (server) code example

```
include "console.iol"
include "math.iol"
include "time.iol"

execution{concurrent}

type session: undefined
type a1: undefined
type a2: undefined
type a3: undefined

cset{
    id: session.id
    a1.id
    a2.id
    a3.id
}


inputPort SD{
    Location:"socket://localhost:8000"
    Protocol: sodep
```

```
23      OneWay: alert1(a1), alert2(a2), alert3(a3), start(session)
24  }
25
26  define printFirst {
27      println@Console(
28          "First alert received "+
29          "from the "+token+" circuit")()
30  }
31
32  main{
33      [start(session)]{
34          println@Console("--- Serving session "+
35              session.id+" ---")();
36          [alert1(a1)]{
37              token="First";
38              printFirst
39          }
40          [alert2(a2)]{
41              token="Second";
42              printFirst
43          }
44          [alert3(a3)]{
45              token="Third";
46              printFirst
47          };
48          println@Console("Alert received")();
49          println@Console("Resetting system...")()
50      }
51  }
```

3.2.4.7 Structured Partial Join

Description

The convergence of *M* branches into a single subsequent branch following a corresponding divergence earlier in the process model. The thread of control is passed to the subsequent branch when *N* of the incoming branches have been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on.

The join construct resets when all active incoming branches have been enabled.

Diagram

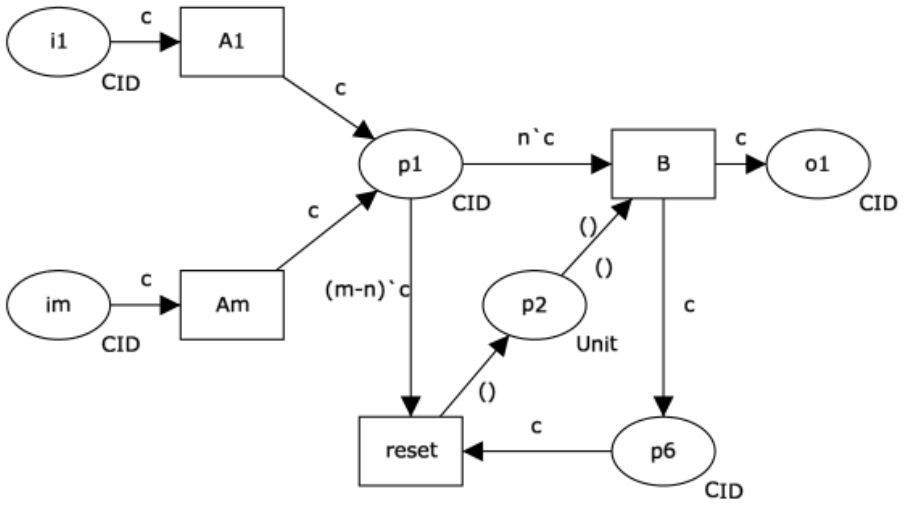


Figure 3.14: Structured Partial Join pattern

Motivation

The *Structured Partial Join* pattern provides a means of merging two or more distinct branches resulting from a specific *Parallel Split* [3.2.3.2] or AND-split construct earlier in a workflow process into a single branch. The join construct does not require triggers on all incoming branches before it can fire. Instead a given

threshold can be defined which describes the circumstances under which the join should fire.

Typically this is presented as the ratio of incoming branches that need to be live for firing as against the total number of incoming branches to the join e.g. a 2-out-of-3 Join signifies that the join construct should fire when two of three incoming arcs are live.

Subsequent completions of other remaining incoming branches have no effect on (and do not trigger) the subsequent branch. As such, the *Structured Partial Join* provides a mechanism for progressing the execution of a process once a specified number of concurrent activities have completed rather than waiting for all of them to complete

JOLIE **Implementation**

The *Structured Partial Joi*n pattern is one possible variant of the *Synchronization* [3.2.3.3] construct, where the number of incoming arcs that will cause the join to fire (N) is between 2 and $M - 1$ (i.e. the total number of incoming branches less one i.e. $2 \leqslant N < M$).

There are a number of possible specializations of the *Synchronization* pattern and they form a hierarchy based on the value of N. Where only one incoming arc must be live for firing (i.e. $N = 1$), this corresponds to one of the variants of the *Discriminator* pattern (3.2.4.4, 3.2.4.5 or 3.2.4.6). An *Synchronization* where all incoming arcs must be live (i.e. $N = M$) is the *Synchronization* or *Generalized AND-Join* [3.2.4.10] pattern.

The pattern provides a means of merging two or more branches in a workflow and progressing execution of the workflow as rapidly as possible by enabling the subsequent (merged) branch as soon as a thread of control has been received on N of the incoming branches where N is less than the total number of incoming branches.

There are two context conditions associated with the use of this pattern:

1. Each of the incoming branches to the join must only be triggered once prior to it being reset;

2. The *Partial Join* resets (and can be re-enabled) once all of its incoming branches have been enabled precisely once.

There are two possible variants on this pattern that arise from relaxing some of the context conditions associated with it. Both of these improve on the efficiency of the join whilst retaining its overall behavior.

The first alternative, the *Blocking Partial Join* [3.2.4.8], removes the requirement that each incoming branch can only be enabled once between join resets. It allows each incoming branch to be triggered multiple times although the construct only resets when one triggering has been received on each input branch.

Second, the *Cancelling Partial Join* [3.2.4.9], improves the efficiency of the pattern further by canceling the other incoming branches to the join construct once N incoming branches have completed.

Both of these alternatives are described and taken into account further.

An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. If there is any ambiguity in how the join condition is specified, an offering is considered to provide partial support for the pattern

The JOLIE code example for this pattern has been written as an evolution of the one used for the *Structured Discriminator* pattern, along with a token based count that releases a `linkOut` variable when the maximum number of token is reached.

As stated in the pattern description, all other waiting operations (branches) must be completed after the `max_token` number is reached. To be able to reset the pattern block, each later operation shall remain in a waiting state until the corresponding client branch reaches the server and release its own `token`. Once all token have been released the block can be reset.

The server part of the *Structured Partial Join* pattern is listed as follows, while for the client part is made reference to the *Standard Discriminator* implementation.

JOLIE code example

Listing 32: Structured Partial Join (server) code example

```
include "console.iol"
include "math.iol"
include "time.iol"

execution{concurrent}

type session: undefined
type a1: undefined
type a2: undefined
type a3: undefined
```

```
11
12  cset{
13      id: session.id
14          a1.id
15          a2.id
16          a3.id
17  }
18
19
20  inputPort SD{
21      Location:"socket://localhost:8000"
22      Protocol: sodep
23      OneWay: alert1(a1), alert2(a2), alert3(a3), start(session)
24  }
25
26  init{
27      max_token=1
28  }
29
30  main{
31      [start(session)]{
32          println@Console("--- Serving session "+
33              session.id+" ---")();
34          token=0;
35          {scope(alert_sensing){
36              {
37                  alert1(a1);
38                  synchronized(lock){
39                  if(token<=max_token){
40                      println@Console(
41                          "a1 took token: "+token)();
42                      println@Console("A1: "+a1.msg)();
43                      if(token==max_token){
44                          linkOut(spj)};
45                      token=token+1
46                  }}
47              }|{
48                  alert2(a2);
49                  synchronized(lock){
50                      if(token<=max_token){
51                          println@Console(
52                              "a2 took token: "+token)();
53                          println@Console("A2: "+a2.msg)();
54                          if(token==max_token){
55                              linkOut(spj)};
56                          token=token+1
57                      }}
58              }|{
59                  alert3(a3);
60                  synchronized(lock){
61                      if(token<=max_token){
62                          println@Console(
63                              "a3 took token: "+token)();
64                          println@Console("A3: "+a3.msg)();
```

```
65                          if(token==max_token){
66                              linkOut(spj)};
67                          token=token+1
68                      }}
69                  }
70          };
71          println@Console("Resetting system...")()}|
72          {linkIn(spj);
73          println@Console("All needed branches "+
74              "reached the system")()}
75      }
76  }
```

3.2.4.8 Blocking Partial Join

Description

The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model.

The thread of control is passed to the subsequent branch when *N* of the incoming branches have been enabled.

The join construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the join has reset.

Diagram

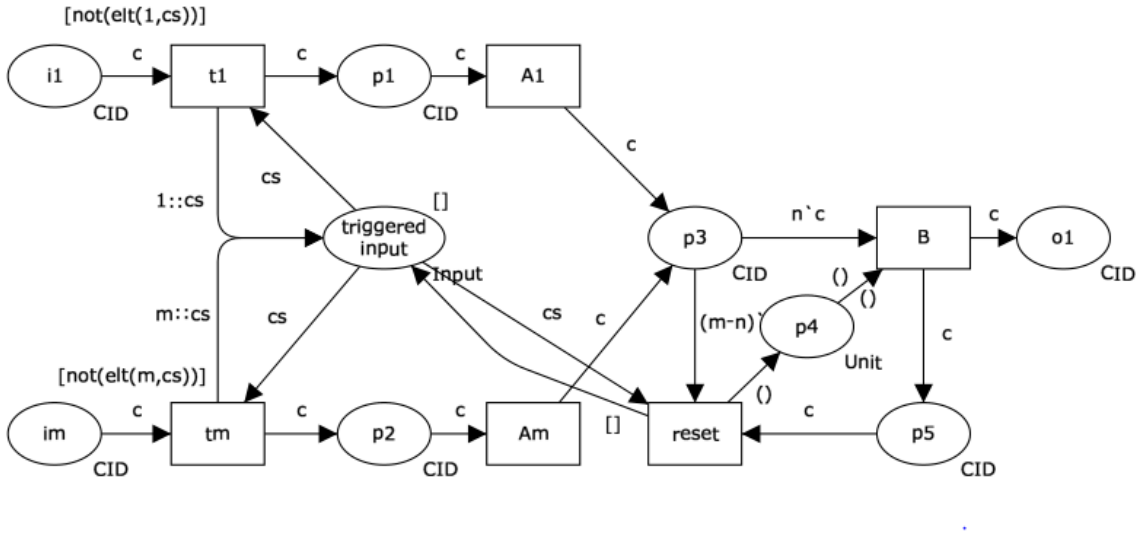


Figure 3.15: Blocking Partial Join pattern

Motivation

The *Blocking Partial Join* is a variant of the *Structured Partial Join* [3.2.4.7] that is able to run in environments where there are concurrent process instances, particularly process instances that have multiple concurrent execution threads.

The *Blocking Partial Join* functions by keeping track of which inputs have been enabled and preventing them from being re-enabled until the construct has reset as a consequence of receiving a trigger on each incoming place.

After N incoming triggers have been received for a given process instance, the join fires. The completion of the remaining $N - M$ branches has no impact on the join except that it is reset when the last of them is received.

The pattern shares the same advantages over the *Structured Partial Join* as the *Blocking Discriminator* [3.2.4.5] does over the *Structured Discriminator* [3.2.4.4], namely greater flexibility as it is able to deal with the situation where a branch is triggered more than once e.g. where the construct exists within a loop and it also shares the same context condition: it can only deal with one case at a time.

JOLIE Implementation

An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. If there is any ambiguity in how the join condition is specified, an offering is considered to provide partial support for the pattern.

The JOLIE code example for this pattern has been written as a slight evolution of the one used for the *Structured Partial Join* pattern which, like the *Blocking Discriminator*, implements its blocking feature using the `execution{sequential}` statement,

The server part of the *Blocking Partial Join* pattern is listed as follows while for the client part is made reference to the *Standard Discriminator* implementation.

JOLIE code example

Listing 33: Blocking Partial Join (server) code example

```
include "console.iol"
include "math.iol"
include "time.iol"

execution{sequential}

type session: undefined
type a1: undefined
type a2: undefined
type a3: undefined

cset{
    id: session.id
        a1.id
        a2.id
```

```
        a3.id
}

inputPort SD{
    Location:"socket://localhost:8000"
    Protocol: sodep
    OneWay: alert1(a1), alert2(a2), alert3(a3), start(session)
}

init{
    max_token=1
}

main{
    [start(session)]{
        println@Console("--- Serving session "+
            session.id+" ---")();
        token=0;
        {scope(alert_sensing){
            {
                alert1(a1);
                synchronized(lock){
                if(token<=max_token){
                    println@Console(
                        "a1 took token: "+token)();
                    println@Console("A1: "+a1.msg)();
                    if(token==max_token){
                        linkOut(spj)};
                    token=token+1
                }}
            }|{
                alert2(a2);
                synchronized(lock){
                    if(token<=max_token){
                        println@Console(
                            "a2 took token: "+token)();
                        println@Console("A2: "+a2.msg)();
                        if(token==max_token){
                            linkOut(spj)};
                        token=token+1
                    }}
            }|{
                alert3(a3);
                synchronized(lock){
                    if(token<=max_token){
                        println@Console(
                            "a3 took token: "+token)();
                        println@Console("A3: "+a3.msg)();
                        if(token==max_token){
                            linkOut(spj)};
                        token=token+1
                }}
```

```
69              }
70          };
71          println@Console("Resetting system...")()}|
72          {linkIn(spj);
73          println@Console("All needed branches "+
74              "reached the system")()}
75      }
76  }
```

3.2.4.9 Canceling Partial Join

Description

The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when N of the incoming branches have been enabled.

Triggering the join also cancels the execution of all of the other incoming branches and resets the construct.

Diagram

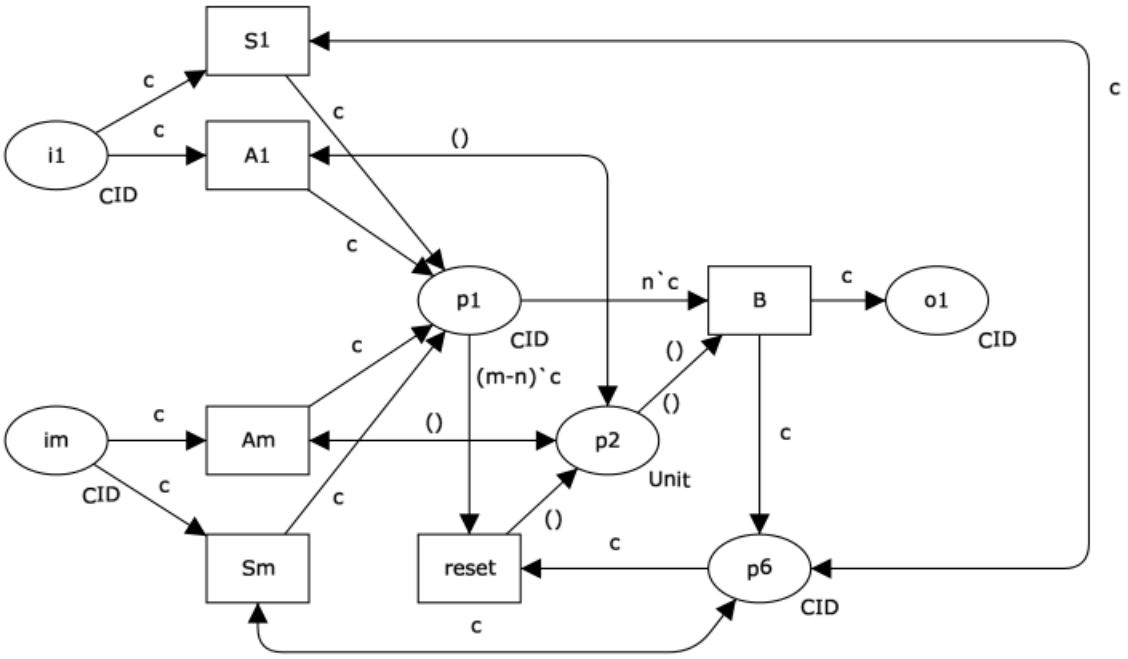


Figure 3.16: Canceling Partial Join pattern

Motivation

The *Canceling Partial Join* pattern provides a means of expediting a process instance where a series of incoming branches to a join need to be synchronized but

only a subset of those activities associated with each of the branches needs to be completed.

It is a context condition of this pattern that only one thread of execution is active for a given process instance in each of the preceding branches to the discriminator. If this is not the case, then the behavior of the process instance is likely to become unpredictable at a later stage during execution.

JOLIE **Implementation**

An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. An offering is considered to provide partial support for the pattern if there are undesirable side-effects associated with the construct firing (e.g. activities in incoming branches which have not completed being recorded as complete or if the semantics associated with the join condition are unclear).

Many of the advanced synchronization patterns assume a safe context (i.e. a place cannot be marked twice for the same process instance). The following pattern is not predicated on this assumption.

The JOLIE code example for this pattern has been written as a slight evolution of the one used for the *Structured Partial Join* pattern which implements its canceling feature using the `throw(kill_session)` statement when the need number of branches reach the block.

The server part of the *Canceling Partial Join* pattern is listed as follows, while for the client part is made reference to the *Standard Discriminator* implementation.

JOLIE **code example**

Listing 34: Canceling Partial Join (server) code example

```
include "console.iol"
include "math.iol"
include "time.iol"

execution{concurrent}

type session: undefined
type a1: undefined
type a2: undefined
type a3: undefined

cset{
    id: session.id
        a1.id
```

```
        a2.id
        a3.id
}

inputPort SD{
    Location:"socket://localhost:8000"
    Protocol: sodep
    OneWay: alert1(a1), alert2(a2), alert3(a3), start(session)
}

init{
    max_token=1
}

main{
    [start(session)]{
        println@Console("--- Serving session "+
            session.id+" ---")();
        token=0;
        install(kill_session=>
            println@Console("All needed branches "+
                "reached the system")();
            println@Console("Resetting system...")());
        scope(alert_sensing){
        {
            alert1(a1);
            synchronized(lock){
            if(token<=max_token){
                println@Console(
                    "a1 took token: "+token)();
                println@Console("A1: "+a1.msg)();
                if(token==max_token){
                    throw(kill_session)};
                token=token+1
            }}
        }|{
            alert2(a2);
            synchronized(lock){
                if(token<=max_token){
                    println@Console(
                        "a2 took token: "+token)();
                    println@Console("A2: "+a2.msg)();
                    if(token==max_token){
                        throw(kill_session)};
                    token=token+1
                }}
        }|{
            alert3(a3);
            synchronized(lock){
                if(token<=max_token){
                    println@Console(
                        "a3 took token: "+token)();
```

```
68                      println@Console("A3: "+a3.msg)();
69                      if(token==max_token){
70                          throw(kill_session)};
71                      token=token+1
72                  }}
73              }
74          }
75      }
76  }
```

3.2.4.10 Generalized AND-Join

Description

The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled.

Additional triggers received on one or more branches between firings of the join persist and are retained for future firings.

Diagram

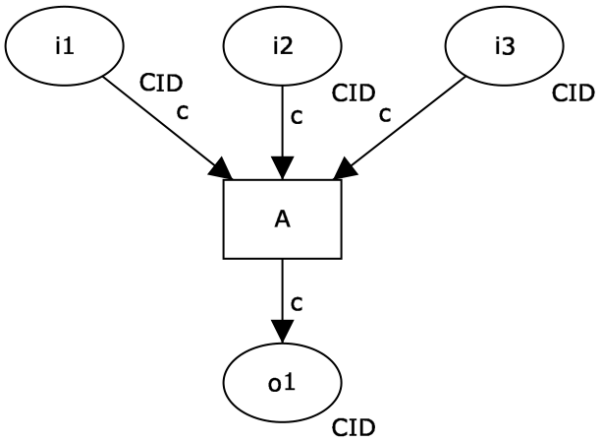


Figure 3.17: AND-Join pattern

Motivation

The *Generalized AND-Join* corresponds to one of the generally accepted notions of the *Synchronization* [3.2.3.3] pattern in which several paths of execution are synchronized and merged together.

Unlike the Synchronization pattern, it supports the situation where one or more incoming branches may receive multiple triggers for the same process instance (i.e. a non-safe context).

JOLIE **Implementation**

The operation of the *Generalized AND-Join* consists of two steps:

1. Before the pattern transition can be enabled, an input token (corresponding to the same case) is required in each of the incoming branches.
2. When there are corresponding tokens in each place, the pattern transition is enabled and consumes a token from each input place.

If there is more than one token at an input place, it ignores additional tokens and they are left in place.

The process analogy to this sequence of events is that the *Synchronization* pattern only fires when a trigger has been received on each incoming branch for a given process instance however additional triggers are retained for future firings.

This approach to *Synchronization* implementation relaxes the context condition associated with the *Synchronization* pattern that only allows it to receive one trigger on each incoming branch and as a result, it is able to be used in concurrent execution environments such as process models which involve loops as well as offerings that do not assume a safe execution environment.

One consideration associated with the *Generalized AND-Join* is that over time, each of the incoming branches should deliver the same number of triggers to the *Synchronization* construct. If this is not the case, then there is the potential for deadlocks to occur and/or tokens to remain after execution has completed.

An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. If there is any ambiguity associated with the specification or use of the construct, an offering is considered to provide partial support for the pattern.

The JOLIE code example for this pattern has been written as a slight evolution of the one used for the *Structured Discriminator* pattern [3.2.4.4] which keeps a concurrent behavior along with a non-deterministic choice approach. While each session is a different one, using several general-scope stacks, each incoming branch (token) is accumulated until their number is sufficient for block output.

The server part of the *Generalized AND-Join* pattern is listed as follows, while for the client part is made reference to the *Standard Discriminator* implementation (session management features taken away).

JOLIE **code example**

Listing 35: Generalized AND-Join (server) code example

```
include "console.iol"
include "math.iol"
include "time.iol"

execution{concurrent}

inputPort SD{
    Location:"socket://localhost:8000"
    Protocol: sodep
    OneWay: alert1, alert2, alert3
}

define addA1 {
    stack.a1=stack.a1+1
}

define addA2{
    stack.a2=stack.a2+1
}

define addA3{
    stack.a3=stack.a3+1
}

define stackCheck{
    if (stack.a1>0 && stack.a2>0 &&stack.a3>0){
        stack.a1=stack.a1-1;
        stack.a2=stack.a2-1;
        stack.a3=stack.a3-1;
        println@Console("All alarms arrived")()
    }
}

init{
    stack->global.stack
}

main{
    [alert1(a1)]{
        synchronized(lock){
            addA1;
            stackCheck
        }
    }
    [alert2(a2)]{
        synchronized(lock){
            addA2;
            stackCheck
        }
    }
    [alert3(a3)]{
        synchronized(lock){
```

```
53                  addA3;
54                  stackCheck
55              }
56          }
57      }
```

3.2.4.11 Local Synchronizing Merge

Description

The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled.

Determination of how many branches require synchronization is made on the basis on information locally available to the merge construct. This may be communicated directly to the merge by the preceding diverging construct or alternatively it can be determined on the basis of local data such as the threads of control arriving at the merge.

Diagram

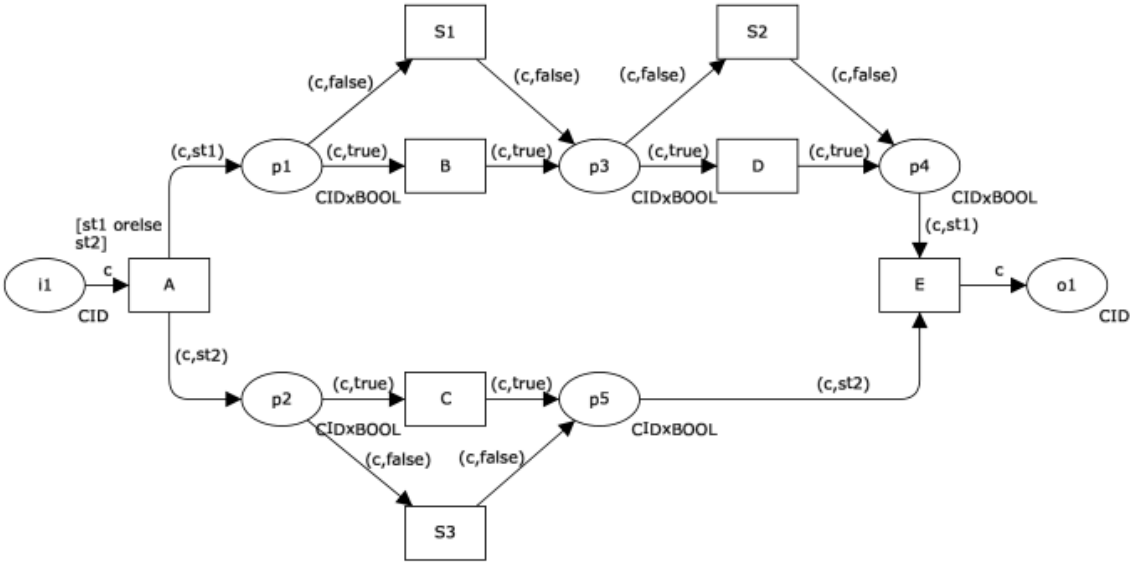


Figure 3.18: Local Synchronizing Merge pattern

Motivation

The *Local Synchronizing Merge* provides a deterministic semantics for the synchronizing merge which does not rely on the process model being structured (as is required for the *Structured Synchronizing Merge*) but also does not require the use of non-local semantics in evaluating when the merge can fire.

JOLIE **Implementation**

The pattern is based on the use of "true" and "false" tokens which are used to indicate whether a branch is enabled or not.

After the divergence (*Parallel Split*), one or both of the outgoing branches may be enabled. The determinant of whether the branch is enabled is that the token passed to the branch contains both the case id as well as a Boolean variable which is "true" if the tasks in the branch are to be executed, "false" otherwise.

As the control-flow token is passed down a branch, if it is a "true" token, then each task that receives the thread of control is executed, otherwise it is skipped.

The *Local Synchronizing Merge* can be evaluated when every incoming branch has delivered a token to the input places for the same case.

There are two context conditions associated with the use of this pattern:

1. once the *Local Synchronizing Merge* has been activated and has not yet been reset, it is not possible for another signal to be received on the activated branch or for multiple signals to be received on any incoming branch, i.e. all input places to the *Local Synchronizing Merge* are safe;

2. the *Local Synchronizing Merge* construct must be able to determine how many incoming branches require synchronization based on local knowledge available to it during execution.

Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumptions. If there is any ambiguity as to the manner in which the synchronization condition is specified, then it rates as partial support.

The JOLIE code example for this pattern has been written as a slight evolution of the one used for the *Synchronization* pattern [3.2.3.3] which implements a local (internal) decision block (deferred choice) whether to wait for `pid2` operation (for synchronization purposes) or not.

The server part of the *Local Synchronizing Merge* pattern is listed as follows, while for the client part is made reference (with merely operational modifications) to the *Synchronization* pattern implementation.

JOLIE **code example**

Listing 36: Local Synchronizing Merge (server) code example

```
include "console.iol"
include "math.iol"

inputPort SyncSvr{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: pid1, pid2
}

main{
    scope(pid1){
        pid1(pid1);
        linkOut(idP1)
    }
    |
    scope(pid2){
        random@Math()(value);
        value=int(2.0*value);
        if(value>0){
            println@Console("pid2 enabled")();
            pid2(pid2);
            linkOut(idP2)
        }
        else{linkOut(idP2)}
    }
    |
    scope(sync){
        {
        linkIn(idP1);
        if(is_defined(pid1)){
        println@Console("pid1 state active, msg: "+pid1.msg)()}
        }|{
        linkIn(idP2);
        if(is_defined(pid2)){
        println@Console("pid2 state active, msg: "+pid2.msg)()}
    }};
    println@Console(
        "Synchornization successfull")()
}
```

3.2.4.12 General Synchronizing Merge

Description

The convergence of two or more branches which diverged earlier in the process into a single subsequent branch.

The thread of control is passed to the subsequent branch when each active incoming branch has been enabled or it is not possible that the branch will be enabled at any future time.

Diagram

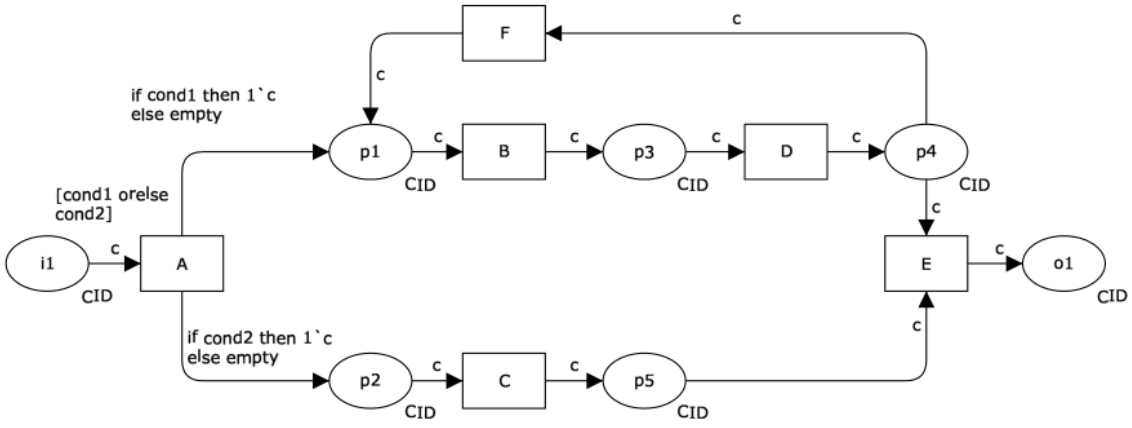


Figure 3.19: General Synchronizing Merge pattern

Motivation

The *General Synchronizing Pattern* pattern provides a general approach to the evaluation of the *Simple Merge* [3.2.3.5] construct in workflow. It is able to be used in non-structured and highly concurrent workflow, including process models that include looping structures.

JOLIE Implementation

The difficulty in implementing the *General Synchronizing Merge* stems from the fact that its evaluation relies on non-local semantics in order to determine when

it can fire. In fact it is easy to see that this construct can lead to the "vicious circle paradox" where two *Simple Merge* blocks depend on one another.

The *Simple Merge* can only be enabled when the thread of control has been received from all incoming branches and it is certain that the remaining incoming branches which have not been enabled will never be enabled at any future time. Determination of this fact requires a (computationally expensive) evaluation of possible future states for the current process instance.

An offering achieves full support if it provides a construct that implements the context requirements for the pattern.

The JOLIE code example for this pattern has been written as a "merge" between the code examples from the *Local Synchronizing Merge* pattern [3.2.4.11] and the *Simple Merge* pattern [3.2.3.5].

The code implements both a local (internal) decision block (deferred choice) and a *Simple Merge* (XOR) block shared between client and server sides, whose behavior is achieved employing a RequestResponse operation between the sides.

The server and client sides of the *General Synchronizing Merge* pattern are listed as follows.

JOLIE code example

Listing 37: General Synchronizing Merge (server) code example

```
include "console.iol"
include "math.iol"

inputPort SyncSvr{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: pid1
    RequestResponse: pid2
}

main{
    min_taxable=150;
    tot_income=0;
    scope(pid1){
        pid1(pid1);
        linkOut(idP1)
    }
    |
    scope(pid2){
        random@Math()(earnings_count);
        earning_days=int(earnings_count*4)+1;
```

```
        for(i=0,i<earning_days,i++){
            pid2(pid2)(pid2){
                tot_income=tot_income+pid2.income;
                println@Console("Received: "+tot_income)();
                if(i<earning_days-1){pid2.needNext=true}
                else{pid2.needNext=false}
            }
        };
        if (tot_income<min_taxable) {
            undef(tot_income)
        };
        linkOut(idP2)
    }|
    scope(sync){
        {linkIn(idP1)|linkIn(idP2)};
        println@Console("Received \""+pid1.income_prov+
            "\" earnings data.")();
        if(is_defined(tot_income)){
        println@Console("Total income: "+tot_income)();
        println@Console("taxted at :"+pid1.income_tax_rate)();
        println@Console("Total taxed: "+double(tot_income)*
            pid1.income_tax_rate)()
        }
        else{
            println@Console("Total earnings are below"+
            " minimum taxable amount.")()}
    };
    println@Console(
        "Synchornization successfull")()
}
```

Listing 38: General Synchronizing Merge (server) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

outputPort sync_svr{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: pid1
    RequestResponse: pid2
}

main{
    scope(pid1){
        sleep@Time(2000)();
        pid1.income_prov="Rented assets";
        pid1.income_tax_rate=0.2;
        pid1@sync_svr(pid1)
```

```
18      }
19      |
20      scope(pid2){
21          sleep@Time(1000)();
22          pid2.needNext=true;
23          while(pid2.needNext){
24              random@Math()(pid2.income);
25              pid2.income=int(pid2.income*100);
26              println@Console("Sending: "+pid2.income)();
27              pid2@sync_svr(pid2)(pid2)
28          }
29      }
30  }
```

3.2.4.13 Thread Merge

Description

At a given point in a process, a nominated number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution.

Diagram

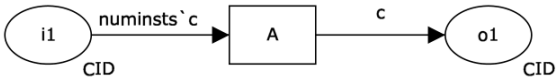


Figure 3.20: Thread Merge pattern

Motivation

This pattern provides a means of merging multiple threads within a given process instance. It is a counterpart to the *Thread Split* pattern [3.2.4.14] which creates multiple execution threads along the same branch. In some situations, it can also be used in conjunction with the *Multiple Instances without Synchronization* pattern [3.2.5.1], however there is the requirement that each of the multiple instances execute along the same branch in the process.

JOLIE **Implementation**

There are two context considerations for this pattern:

1. the number of threads needing to be merged must be known at design-time;
2. only execution threads for the same process instance can be merged. If the pattern is used to merge independent execution threads arising from some form of activity spawning, then it must be possible to identify the specific threads that need to be coalesced.

An offering achieves full support for this pattern if it provides a construct that satisfies the context requirements. If any degree of programmatic extension is required to achieve the same behavior, then the partial support rating applies.

The JOLIE code example for this pattern has been written to be in compliance with the second context condition too. Even for this simple pattern a server-client approach is used such that employing the `cset` and concurrent execution features make possible having several separated branches running simultaneously while is possible to clearly identifying each thread's provenience.

The server and client sides of the *Thread Merge* pattern are listed as follows.

JOLIE **code example**

Listing 39: Thread Merge (server) code example

```
include "console.iol"

cset{
    branch_id: msg.branch_id
}

type msg: undefined

inputPort TM{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: start(msg), tm(msg)
}

execution{concurrent}

main{
    [start(msg)]{
        tc=0;
        while(tc<10){
            tm(msg);
            tc++;
            println@Console("Received thread n."+tc+
                " from bid: "+msg.branch_id)()
        };
        println@Console("All thread received"+
            " from bid: "+msg.branch_id+
            ", starting finishing procedure.")()
    }
}
```

Listing 40: Thread Merge (server) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

outputPort TM{
```

```
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: start, tm
}

main{
    random@Math()(branch_id);
    msg.branch_id=int(branch_id*1000);
    start@TM(msg);
    for(i=0,i<10,i++){
        random@Math()(exec_time);
        exec_time=int(exec_time*10000);
        sleep@Time(exec_time)();
        tm@TM(msg)
    }
}
```

3.2.4.14 Thread Split

Description

At a given point in a process, a nominated number of execution threads can be initiated in a single branch of the same process instance.

Diagram

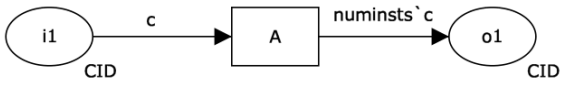


Figure 3.21: Thread Split pattern

Motivation

This pattern provides a means of triggering multiple execution threads along a branch within a given process instance. It is a counterpart to the *Thread Merge* pattern [3.2.4.13] which merges multiple execution threads along the same branch. Unless used in conjunction with the *Thread Merge* pattern, the execution threads will run independently to the end of the process.

JOLIE **Implementation**

There are two context considerations for this pattern:

1. the number of threads to be initiated must be known at design-time
2. all threads must be initiated from the same point in the process model (i.e. they must flow along the same branch).

An offering achieves full support for this pattern if it provides a construct that satisfies the context requirements. If any degree of programmatic extension is required to achieve the same behavior, then the partial support rating applies.

The JOLIE code example for this pattern has been written with a server-client approach taking advantage of non-deterministic choice feature to run each thread independently.

The server side of the code example of the *Thread Split* pattern is listed as follows, since the client implementation consists of a single `for` block invoking ten times the `ts` operation at server side .

JOLIE **code example**

Listing 41: Thread Split (server) code example

```
include "console.iol"
include "time.iol"
include "math.iol"


inputPort TS{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: ts
}

execution{concurrent}

main{
    [ts(msg)]{
        random@Math()(tid);
        tid=int(tid*1000);
        for(i=1,i<4,i++){
            random@Math()(exec_time);
            exec_time=int(exec_time*10000);
            sleep@Time(exec_time)();
            println@Console("Thread: "+tid+
                " rep: "+i)()
        }
    }
}
```

3.2.5 Multiple Instance Patterns

Multiple instance patterns describe situations where there are multiple threads of execution active in a process model which relate to the same activity (and hence share the same implementation definition).

Multiple instances can arise in three situations:

- An activity is able to initiate multiple instances of itself when triggered (namely a multiple instance activity);
- A given activity is initiated multiple times as a consequence of it receiving several independent triggerings;
- Two or more activities in a process share the same implementation definition. This may be the same activity definition in the case of a multiple instance activity or a common sub-process definition in the case of a block activity. Two (or more) of these activities are triggered such that their executions overlap (either partially or wholly).

Although all of these situations potentially involve multiple concurrent instances of an activity or sub-process, it is the first of them in which we are most interested as they require the triggering and synchronization of multiple concurrent activity instances. This group of patterns focuses on the various ways in which these events can occur.

3.2.5.1 Multiple Instances without Synchronization

Description

Within a given process instance, multiple instances of an activity can be created. These instances are independent of each other and run concurrently. There is no requirement to synchronize them upon completion.

Diagram

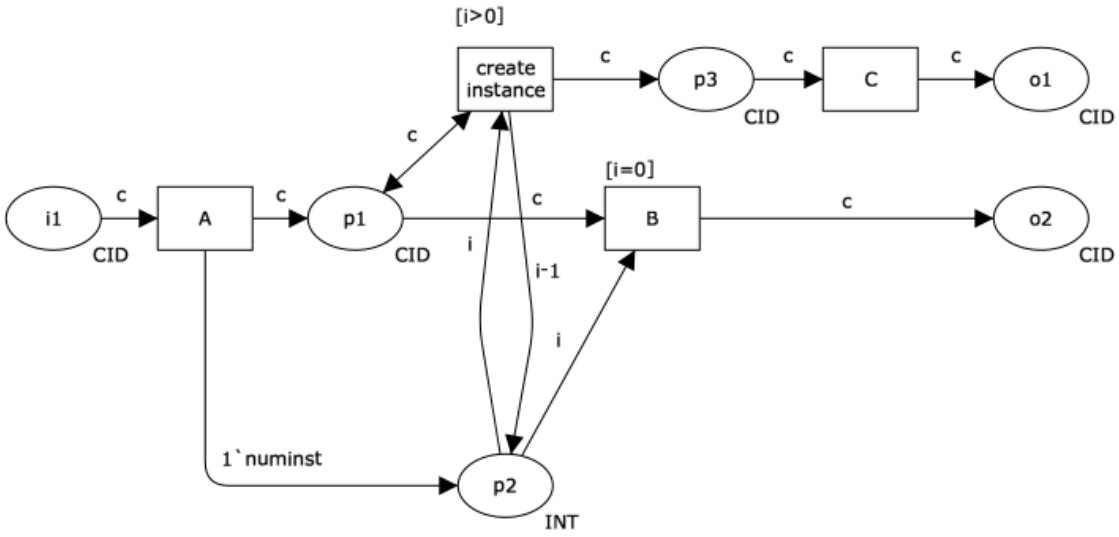


Figure 3.22: Multiple Instances without Synchronization pattern

Motivation

This pattern provides a means of creating multiple instances of a given activity. It is particularly suited to situations where the number of individual activities required is known before the spawning action commences, the activities can execute independently of each other and no subsequent synchronization is required.

JOLIE **Implementation**

There are two context conditions associated with this pattern:

1. each of the multiple instance activities that are created must execute within the context of the process instance from which they were started (i.e. they must share the same case id and have access to the same data elements);

2. each of the multiple instance activities must execute independently from and without reference to the activity that started them.

Where an offering provides support for this pattern, one issue that can potentially arise is how the various threads of execution might be synchronized at some future point in the process. This is potentially problematic as it is likely that the individual threads of execution may ultimately flow down the same path. In recognition of this need, the *Thread Merge* [3.2.4.13] and *Thread Split* [3.2.4.14] patterns have been introduced.

An offering achieves full support if it satisfies the context requirements for the pattern. Where the newly created activity instances run in a distinct process instance to the activity that started them or they cannot access the same data elements as the parent activity, the offering achieves only partial support.

The JOLIE code example for this pattern is very recurrent in multi-threaded/concurrent implementation provided for previous patterns, a straightforward example of this pattern implementation can be found in *Thread Split* code example where multiple instances of the same activity can be run simultaneously, each one with it's reserved context.

3.2.5.2 Multiple Instances with *a priori* Design-Time Knowledge

Description

Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently.

It is necessary to synchronize the activity instances at completion before any subsequent activities can be triggered.

Diagram

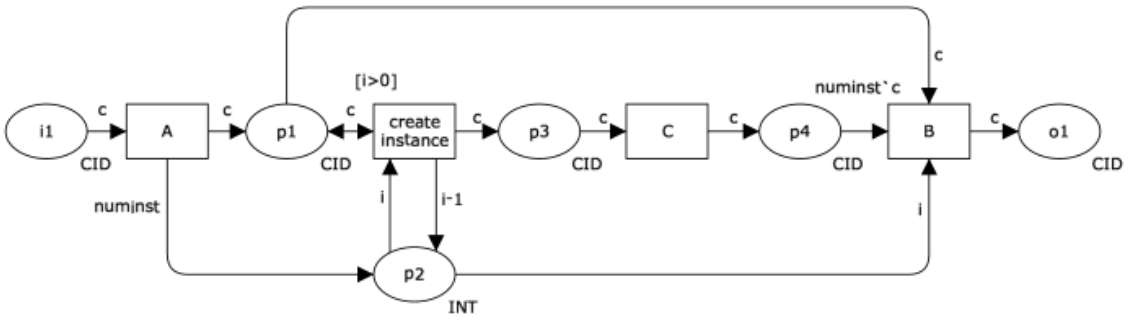


Figure 3.23: Multiple Instances with a priori Design-Time Knowledge pattern

Motivation

This pattern provides the basis for concurrent execution of a nominated activity a predefined number of times. It also ensures that all activity instances are complete before subsequent activities are initiated.

JOLIE **Implementation**

There are three context conditions associated with this pattern:

1. the number of activity instances required must be specified in the design-time process model;

2. it must be possible for the activity instances to execute concurrently (although it is not necessarily required that they do all execute in parallel);

3. all activity instances must complete before subsequent activities in the process can be triggered.

Many offerings provide a work-around for this pattern by embedding some form of activity invocation within a loop. These implementation approaches have two significant problems associated with them:

1. the activity invocations occur at discrete time intervals and it is possible for the individual activity instances to have potentially distinct states at the time they are invoked (i.e. the activities do not need to be executed in sequence and can be handled concurrently)

2. there is no consideration of the means by which the distinct activity instances will be synchronized.

These issues, together with the necessity for the designer to effectively craft the pattern themselves (rather than having it provided by the offering) rule out this form of implementation from being considered as satisfying the requirements for full support.

An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern. Although work-arounds are possible which achieve the same behavior through the use of various constructs within an offering such as activity replication or loops, they have a number of shortcomings and are not considered to constitute support for the pattern.

The JOLIE code example for this pattern is very recurrent too [3.2.5.1] in multi-threaded/concurrent implementation provided for previous patterns examples, a straightforward example of this pattern implementation can be found in *: Structured Synchronizing Merge* [3.2.4.2] code example where multiple instances, defined at design-time, run simultaneously, while a synchronizing block is used to merge all branches after their completion.

3.2.5.3 Multiple Instances with *a priori* Run-Time Knowledge

Within a given process instance, multiple instances of an activity can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but is known before the activity instances must be created.

Once initiated, these instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent activities can be triggered.

Diagram

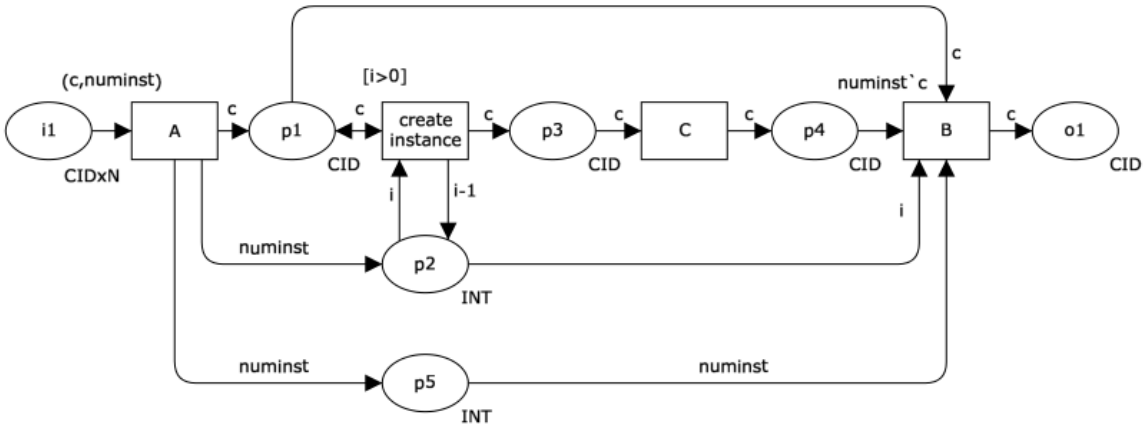


Figure 3.24: Multiple Instances with a priori Run-Time Knowledge pattern

Motivation

The *Multiple Instances with a priori Run-Time Knowledge* pattern provides a means of executing multiple instances of a given activity in a synchronized manner with the determination of exactly how many instances will be created being deferred to the latest possible time before the first of the activities is started.

JOLIE **Implementation**

There are three context conditions associated with this pattern:

1. the number of activity instances required must be known at run-time prior to the invocation of the multiple instance activity;
2. it must be possible for the activity instances to execute concurrently (although it is not necessarily required that they do all execute in parallel);
3. all activity instances must complete before subsequent activities in the process can be triggered.

An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern.

The JOLIE code example for this pattern can be found in *General Synchronizing Merge* [3.2.4.12] code example where multiple instances, defined at run_time according to a random counter, run sequentially, while a synchronizing block is used to merge all branches after their completion.

3.2.5.4 Multiple Instances without *a priori* Run-Time Knowledge

Within a given process instance, multiple instances of an activity can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed.

Once initiated, these instances are independent of each other and run concurrently.

At any time, whilst instances are running, it is possible for additional instances to be initiated. It is necessary to synchronize the instances at completion before any subsequent activities can be triggered.

Diagram

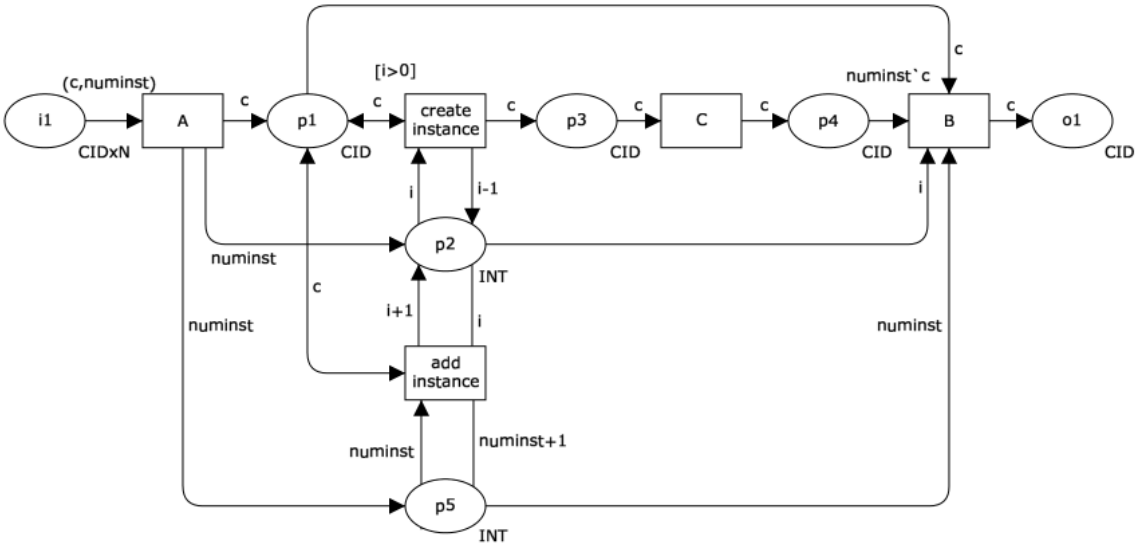


Figure 3.25: Multiple Instances without a priori Run-Time Knowledge pattern

Motivation

This pattern is an extension to the *Multiple Instances with a priori Run-Time Knowledge* pattern [3.2.5.3] which defers the need to determine how many concurrent

instances of the activity are required until the last possible moment – either when the final join construct fires or the last of the executing instances completes.

It offers more flexibility in that additional instances can be created "on-the-fly" without any necessary change to the process model or the synchronization conditions for the activity.

JOLIE **Implementation**

There are three context conditions associated with this pattern:

1. the number of activity instances required must be known at run-time prior to the completion of the multiple instance activity (it's worth noting that the final number of instances does not need to be known when initializing the multiple instance activity);

2. it must be possible for the activity instances to execute concurrently (although it is not necessarily required that they do all execute in parallel);

3. all activity instances must complete before subsequent activities in the process can be triggered.

An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern.

This pattern rises some implementation (and behavioral) concerns in JOLIE.

Both implementation, as a single non-communicating execution and with the traditional server-client approach, can't fully support of the *Multiple Instances without a priori Run-Time Knowledge* pattern requirement because of thread execution modality implemented by JOLIE.

In JOLIE parallel threads can run simultaneously to each other only if programmatically defined (i.e. at design time), thus only a predetermined number of branches can potentially run in parallel, while each thread runs sequentially within the same branch.This limitation in threads execution (and the absence of recursive constructs) makes impossible to fully implement the pattern mechanism.

The code example listed as follows implements a main process branch whose number of threads is known at design-time, along with a parallel `add_thread` functionality which, once invoked, executes a single parallel (side) thread, which is added to the total thread count, to be synchronized with the ones of the main

branch. The implementation issue raised is that no additional parallel side threads can be invoked next to the one described before, thus forbidding other concurrent thread invocations to be received and run.

Considering an execution example, while the main process (`start`) branch has been invoked and running, the `add_thread` operation is invoked to run a side thread, which keeps the `add_thread` process busy for several seconds before letting it to reset and to able to receive another `add_thread` request. For this reason, even if the client asks for N additional threads to be run, the server would probably catch only a subset M ($M < N$) of those because, although concurrent `add_thread` invocations are run, the corresponding server-side operation is busy, running a single thread, resulting in missing invocations and less-than-expected threads to be run.

The server and client sides of the code example of the *Multiple Instances without a priori Run-Time Knowledge* pattern are listed as follows.

JOLIE **code example**

Listing 42: Multiple Instances without a priori Run-Time Knowledge (server) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

inputPort MIWAPRTK{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: start, add_thread
}

main{
    completed_threads=0;
    runnable_threads=0;
    install(process_complete=>
        println@Console("All "+completed_threads+
            " threads executed.")());
    {//MAIN PROCESS SEQUENCE
        start(msg);
        runnable_threads=msg.td_count;
        for(i=0,i<msg.td_count,i++){
            random@Math()(exec_time);
            exec_time=int(exec_time*3000);
            sleep@Time(exec_time)();
            synchronized(lock){completed_threads=
                completed_threads+1};
```

```
            println@Console("Completed main thread "+(i+1)+
                ", "+(runnable_threads-completed_threads)+
                " still running." )();
            if(completed_threads==runnable_threads){
                throw(process_complete)
            }
        }
    }|
    {//SIDE SEQUENCE
        while(true){
            if(runnable_threads>0){
                add_thread(msg);
                synchronized(lock){runnable_threads++};
                sleep@Time(msg.td_exec_time)();
                println@Console("Completed side thread, "+
                    (runnable_threads-completed_threads)+
                    " still running." )();
                synchronized(lock){completed_threads=
                    completed_threads+1};
                if(completed_threads==runnable_threads){
                    throw(process_complete)
                }
            }
        }
    }
}
```

Listing 43: Multiple Instances without a priori Run-Time Knowledge (client) code example

```
include "console.iol"
include "time.iol"
include "math.iol"


outputPort MIWAPRTK{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: start, add_thread
}

main{
    {
        random@Math()(msg.td_count);
        msg.td_count=int(msg.td_count*10);
        println@Console("Started main "+msg.td_count+
            " threads block.")();
        start@MIWAPRTK(msg)
    }|
    {
```

```
21          while(true){
22              random@Math()(msg.exec_time);
23              msg.exec_time=int(msg.exec_time*5000);
24              sleep@Time(msg.exec_time)();
25              println@Console("Adding side thread.")();
26              add_thread@MIWAPRTK(msg)
27          }
28      }
29  }
```

3.2.5.5 Static Partial Join for Multiple Instances

Within a given process instance, multiple concurrent instances of an activity can be created. The required number of instances is known when the first activity instance commences. Once N of the activity instances have completed, the next activity in the process is triggered. Subsequent completions of the remaining $M - N$ instances are inconsequential.

Diagram

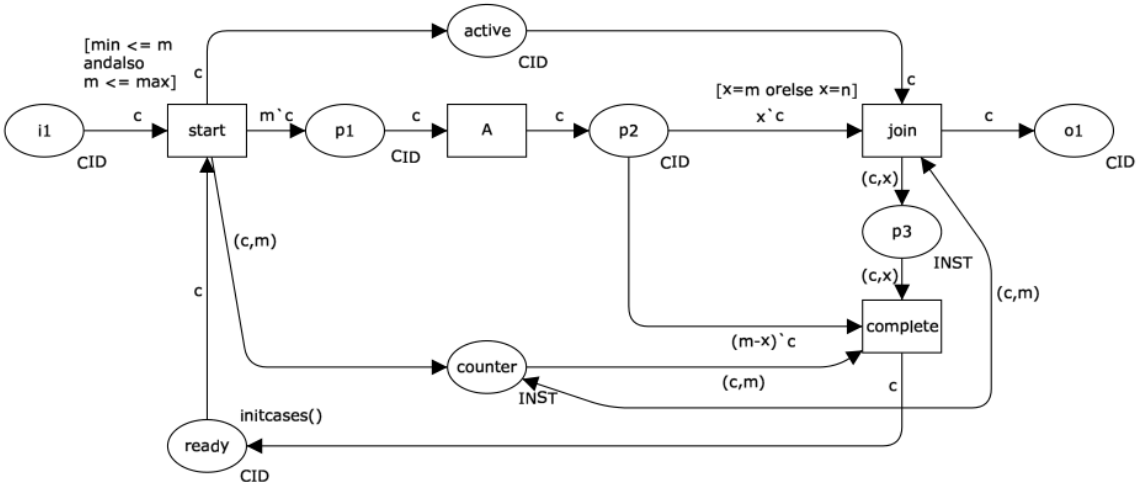


Figure 3.26: Static Partial Join for Multiple Instances pattern

Motivation

The *Static Partial Join for Multiple Instances* pattern is an extension to the *Multiple Instances with a priori Run-time Knowledge* pattern [3.2.5.3] which allows the process instance to continue once a given number of the activity instances have completed rather than requiring all of them to finish before the subsequent activity can be triggered.

JOLIE Implementation

The general format of the *Static Partial Join for Multiple Instances* defines several context conditions associated with it:

- the number of concurrent activity instances is known prior to activity commencement;
- the number of activities that need to completed before subsequent activities in the process model can be triggered is known prior to activity commencement;
- once the required number of activities have completed, the thread of control can immediately be passed to subsequent activities;
- the number of instances that must complete for the join to be triggered (N) cannot be greater than the total number of concurrent activity instances (M), i.e. $N \not> M$;
- completion of the remaining activity instances do not trigger a subsequent activity, however all instances must have completed in order for the join construct to reset and be subsequently re-enabled.

An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern. It achieves partial support if there is any ambiguity associated with the specification of the join condition.

The JOLIE implementation of this pattern can be obtained as a slight modification to the *Structured Partial Join* pattern [3.2.4.7] example, where the main difference relies in the client part, where multiple instances of the same client can be run and invoke parallelly the token-release operations.

3.2.5.6 Canceling Partial Join for Multiple Instances

Within a given process instance, multiple concurrent instances of an activity can be created. The required number of instances is known when the first activity instance commences. Once N of the activity instances have completed, the next activity in the process is triggered and the remaining $M - N$ instances are canceled.

Diagram

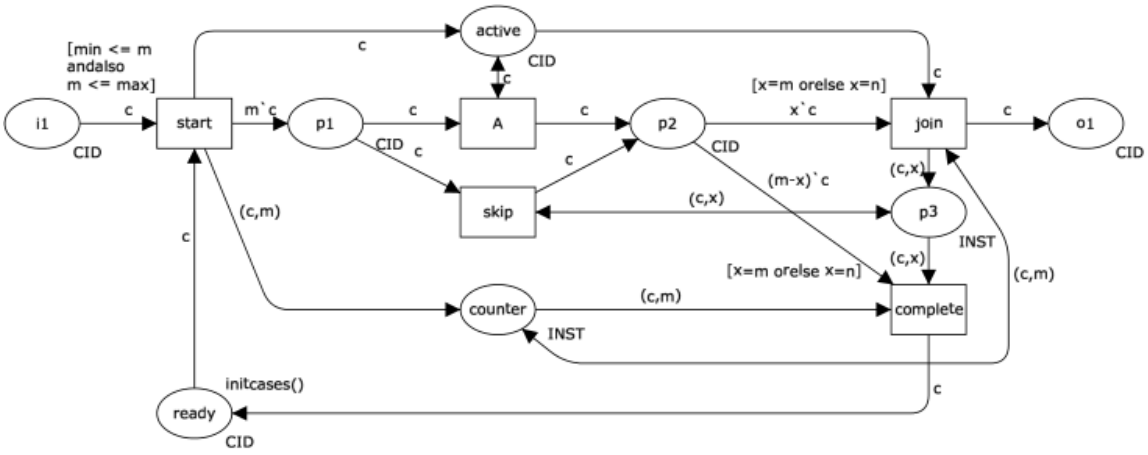


Figure 3.27: Canceling Partial Join for Multiple Instances pattern

Motivation

This pattern provides a variant of the multiple instances pattern that expedites process throughput by both allowing the process to continue to the next activity once a specified number (N) of the multiple instance activities have completed and also cancels any remaining activity instances negating the need to expend any further effort executing them.

JOLIE Implementation

This pattern shares four context conditions with the *Static Partial Join for Multiple Instances* pattern [3.2.5.5]: the number of concurrent activity instances (M) and the

completion threshold (N) must be known before commencement, the number of instances that must complete for the join to be triggered (N) cannot be greater than the total number of concurrent activity instances (M), i.e. $N \not> N$ and subsequent activities can be triggered as soon as the required completion threshold has been reached, however the final context condition is relaxed and the pattern is able to be re-enabled almost immediately after the completion threshold is reached as remaining activity instances are canceled.

An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern. An offering achieves partial support if there is any ambiguity associated with the implementation of the pattern.

The JOLIE implementation of this pattern can be obtained as a slight modification to the *Canceling Partial Join* pattern [3.2.4.9] example, where the main difference relies in the client part, where multiple instances of the same client can be run and invoke parallelly the token-release operations.

3.2.5.7 Dynamic Partial Join for Multiple Instances

Within a given process instance, multiple concurrent instances of an activity can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed.

At any time, whilst instances are running, it is possible for additional instances to be initiated providing the ability to do so has not been disabled. A completion condition is specified which is evaluated each time an instance of the activity completes.

Once the completion condition evaluates to true, the next activity in the process is triggered. Subsequent completions of the remaining activity instances are inconsequential and no new instances can be created.

Diagram

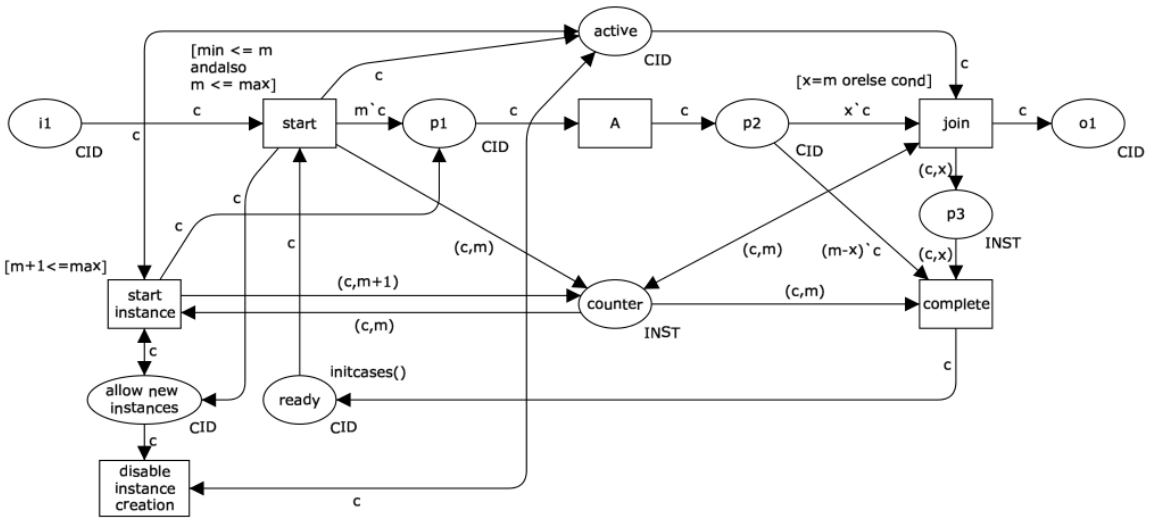


Figure 3.28: Dynamic Partial Join for Multiple Instances pattern

Motivation

This pattern is a variant of the *Multiple Instances without a priori Run-time Knowledge* pattern [3.2.5.4] that allows the thread of execution to pass to subsequent activities once a specified completion condition is met. It allows the process to progress without requiring that all instances associated with a multiple instance activity have completed.

JOLIE Implementation

An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. It achieves partial support if the creation of activity instances cannot be disabled once the first activity instance has commenced.

The JOLIE implementation of this pattern can be obtained as a slight modification to the previous *Multiple Instances without a priori Run-time Knowledge* pattern example, where the main difference relies in the client part, where multiple instances of the same client can be run and invoke parallelly the token-release operations.

JOLIE code example

Listing 44: Dynamic Partial Join for Multiple Instances (server) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

inputPort DPJMI{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: start, add_thread
}

main{
    completed_threads=0;
    runnable_threads=0;
    install(process_complete=>
        println@Console("All "+completed_threads+
            " threads executed.")());
    {//MAIN PROCESS SEQUENCE
        start(msg);
        runnable_threads=msg.max_td;
        for(i=0,i<msg.td_count,i++){
            random@Math()(exec_time);
            exec_time=int(exec_time*3000);
            sleep@Time(exec_time)();
            synchronized(lock){completed_threads=
                completed_threads+1};
            println@Console("Completed main thread "+(i+1)+
                ", "+(runnable_threads-completed_threads)+
```

```
                " still running." )();
            if(completed_threads==runnable_threads){
                throw(process_complete)
            }
        }
    }|
    {//SIDE SEQUENCE
        while(true){
            if(runnable_threads>0){
                add_thread(msg);
                sleep@Time(msg.td_exec_time)();
                println@Console("Completed side thread, "+
                    (runnable_threads-completed_threads)+
                    " still running." )();
                synchronized(lock){completed_threads=
                    completed_threads+1};
                if(completed_threads==runnable_threads){
                    throw(process_complete)
                }
            }
        }
    }
}
```

Listing 45: Dynamic Partial Join for Multiple Instances (client) code example

```
include "console.iol"
include "time.iol"
include "math.iol"


outputPort DPJMI{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: start, add_thread
}

main{
    {
        random@Math()(msg.td_count);
        msg.td_count=int(msg.td_count*10)+1;
        random@Math()(max_percent);
        msg.max_td=int(max_percent*msg.td_count)+1;
        println@Console("Started main "+msg.td_count+
            " threads block. "+msg.max_td+
            " needed before completion.")();
        start@DPJMI(msg)
    }|
    {
        while(true){
```

```
25              random@Math()(msg.exec_time);
26              msg.exec_time=int(msg.exec_time*5000);
27              sleep@Time(msg.exec_time)();
28              println@Console("Adding side thread.")();
29              add_thread@DPJMI(msg)
30          }
31      }
32  }
```

3.2.6 State-based Patterns

State-based patterns reflect situations for which solutions are most easily accomplished in process languages that support the notion of state. In this context, it's considered the state of a process instance to include the broad collection of data associated with current execution including the status of various activities as well as process-relevant working data such as activity and case data elements.

3.2.6.1 Deferred Choice

A point in a workflow process where one of several branches is chosen based on interaction with the operating environment.

Prior to the decision, all branches present possible future courses of execution. The decision is made by initiating the first activity in one of the branches, i.e. there is no explicit choice but rather a race between different branches. After the decision is made, execution alternatives in branches other than the one selected are withdrawn.

Diagram

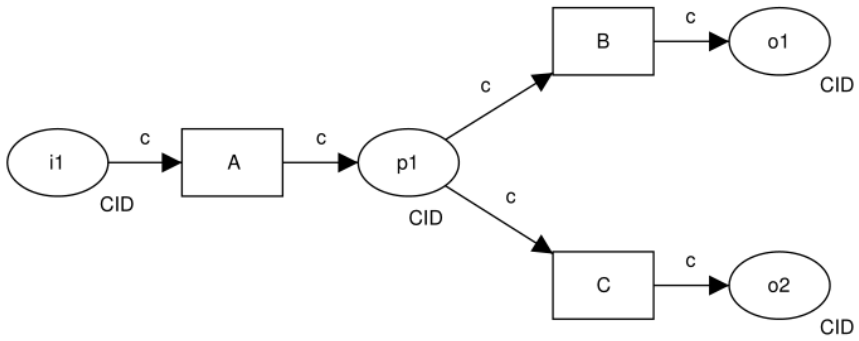


Figure 3.29: Deferred Choice pattern

Motivation

The *Deferred Choice* pattern provides the ability to defer the moment of choice in a process, i.e. the moment as to which one of several possible courses of action should be chosen is delayed to the last possible time and is based on factors external to the process instance (e.g. incoming messages, environment data, resource availability, timeouts etc.).

Up until the point at which the decision is made, any of the alternatives presented represent viable courses of future action.

JOLIE **Implementation**

It is a context condition of this pattern that once one of the possible alternative courses of action is chosen, any possible actions associated with other branches are immediately withdrawn.

An offering achieves full support if it provides a construct that satisfies the context criteria for the pattern. If there are any restrictions on which branches can be selected or withdrawn, then the offering is rated as having partial support.

The JOLIE support of this pattern can be obtained in two ways:

- via the primitive non-deterministic input choice construct, which implements exactly the first-to-reach-XOR mechanism described by this pattern (whose example is not given for the sake of brevity);
- whereas implementations based on environment data, timeouts and resource availability can be obtained by a parallel thread run along with invocation waiting operation which "senses" the execution environment modifications and discards the corresponding subsequent operations.

An example of such an implementation is given as follows.

JOLIE **code example**

Listing 46: Deferred Choice (server) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

inputPort DC{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: pid1, pid2
```

```
}

main{
    pid1.in_time=true;
    pid2.in_time=true;
    {
        pid1(msg);
        if(pid1.in_time){
            pid2.in_time=false;
            println@Console("PID1 executed")()
        }
    }|{
        pid2(msg);
        if(pid2.in_time){
            pid1.in_time=false;
            println@Console("PID2 executed")()
        }
    }|{
        random@Math()(pid1.timeout);
        pid1.timeout=int(pid1.timeout*5000);
        println@Console("PID1 timeout: "+pid1.timeout)();
        sleep@Time(pid1.timeout)();
        synchronized(lock){
            if(pid2.in_time){
                pid1.in_time=false;
                println@Console("PID1 timeout")()
        }}
    }|{
        random@Math()(pid2.timeout);
        pid2.timeout=int(pid2.timeout*5000);
        println@Console("PID2 timeout: "+pid2.timeout)();
        sleep@Time(pid2.timeout)();
        synchronized(lock){
            if(pid1.in_time){
                pid2.in_time=false;
                println@Console("PID2 timed out")()
        }}
    }
}
```

Listing 47: Deferred Choice (client) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

outputPort DC{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: pid1, pid2
```

```
}

main{
    {
        random@Math(pid1.time)();
        pid1.time=int(pid1.time*5000);
        sleep@Time(pid1.time)();
        println@Console("PID1 Started")();
        pid1@DC(msg)
    }|{
        random@Math(pid2.time)();
        pid2.time=int(pid2.time*5000);
        sleep@Time(pid2.time)();
        println@Console("PID2 Started")();
        pid2@DC(msg)
    }
}
```

3.2.6.2 Interleaved Parallel Routing

A set of activities has a partial ordering defining the requirements with respect to the order in which they must be executed. Each activity in the set must be executed once and they can be completed in any order that accords with the partial order.

However, as an additional requirement, no two activities can be executed at the same time (i.e. no two activities can be active for the same process instance at the same time).

Diagram

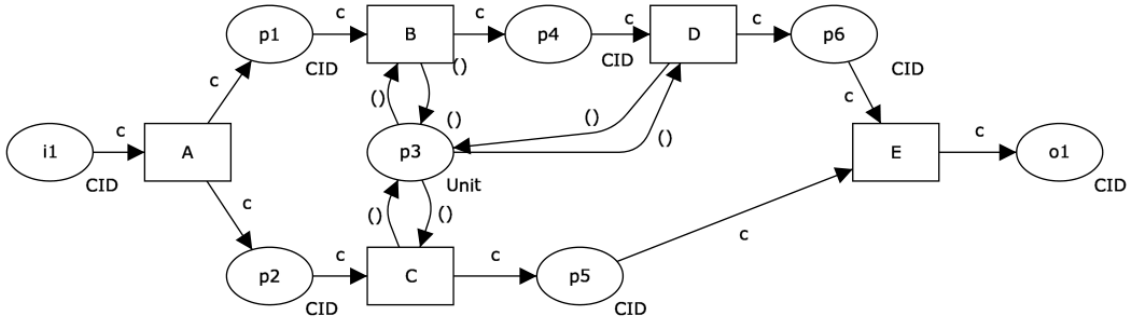


Figure 3.30: Interleaved Parallel Routing pattern

Motivation

The *Interleaved Parallel Routing* pattern offers the possibility of relaxing the strict ordering that a process usually imposes over a set of activities. Note that *Interleaved Parallel Routing* is related to mutual exclusion, i.e. a semaphore makes sure that activities are not executed at the same time without enforcing a particular order.

JOLIE **Implementation**

There are three context conditions associated with this pattern:

1. for a given process instance, no two activities from the set of activities subject to interleaved parallel routing may be executed at the same time,

2. there must be some (partial) ordering defined between the activities
3. activities must be initiated and completed on a sequential basis, it is not possible to suspend one activity during its execution to work on another.

An offering achieves full support if it it is able to satisfy the context criteria for the pattern. It achieves a partial support rating if there are any limitations on the set of activities that be interleaved or if activities can be suspended during execution.

An example of such an implementation is given as follows.

JOLIE **code example**

Listing 48: Interleaved Parallel Routing (server) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

inputPort IPR{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: alarm1, alarm2, alarm3
}

main{
    {{
        synchronized(lock){alarm1(msg);
            println@Console("alarm1 received, "+
                "processing initiated")();
                sleep@Time(2000)();
            println@Console("alarm1 done.")()
        }
    }|
    {
        synchronized(lock){
            alarm2(msg);
            println@Console("alarm2 received, "+
                "processing initiated")();
            sleep@Time(1000)();
            println@Console("alarm2 done.")();
            alarm3(msg);
            println@Console("alarm3 received, "+
                "processing initiated")();
            sleep@Time(1000)();
            println@Console("alarm2 done.")()
        }
    }};
    println@Console("All alarms received.")();
    println@Console("System reset.")()
}
```

Listing 49: Interleaved Parallel Routing (client) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

outputPort IPR{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: alarm1, alarm2, alarm3
}

main{
    {
        random@Math()(a1.s_time);
        sleep@Time(int(a1.s_time*5000))();
        alarm1@IPR(msg);
        println@Console("alarm1 sent.")()
    }|
    {
        random@Math()(a2.s_time);
        sleep@Time(int(a2.s_time*5000))();
        alarm2@IPR(msg);
        println@Console("alarm2 sent.")()
    }|{
        random@Math()(a3.s_time);
        sleep@Time(int(a3.s_time*5000))();
        alarm3@IPR(msg);
        println@Console("alarm3 sent.")()
    }
}
```

3.2.6.3 Milestone

An activity is only enabled when the process instance (of which it is part) is in a specific state (typically in a parallel branch). The state is assumed to be a specific execution point (also known as a milestone) in the process model. When this execution point is reached the nominated activity can be enabled. If the process instance has progressed beyond this state, then the activity cannot be enabled now or at any future time (i.e. the deadline has expired).

Note that the execution does not influence the state itself, i.e. unlike normal control-flow dependencies it is a test rather than a trigger.

Diagram

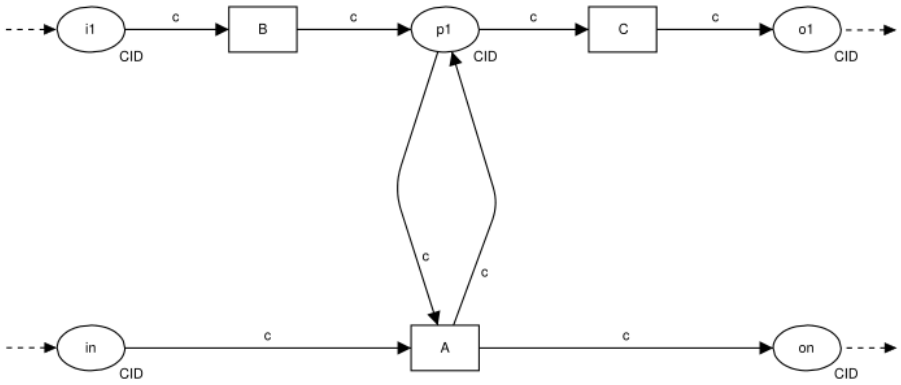


Figure 3.31: Milestone pattern

Motivation

The *Milestone* pattern provides a mechanism for supporting the conditional execution of an activity or sub-process (possibly on a repeated basis) where the process instance is in a given state.

The notion of state is generally taken to mean that control-flow has reached a nominated point in the execution of the process instance (i.e. a milestone). As such, it provides a means of synchronizing two distinct branches of a process instance, such that one branch cannot proceed unless the other branch has reached a specified state.

JOLIE **Implementation**

An offering achieves full support if it provides a construct that allows the execution of a given activity to be dependent on the process instance being in some predefined state.

An example of such an implementation is given as follows, using a `throw` statement, caught by the corresponding `install` statement, that "kills" the sibling OneWay operation and resets the looping procedure. While the server part is reported below, the client one is not listed since it only contains a widely used and common method which waits for a random number of seconds and sends its `alert` message.

JOLIE **code example**

Listing 50: Milestone (server) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

inputPort MILE{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: alert
}

main{
    while(true){
        random@Math()(s_time);
        sleep@Time(int(s_time*5000))();
        println@Console("System Check complete.")();
        scope(myScope){
            install(alert=>
                println@Console("!!!Alert sensed!!!")());
            install(no_alert=>
                println@Console("No alert sensed")();
                throw(killOneWay)
                );
            {
            install(killOneWay=>
                    println@Console("OneWay killed")());
            println@Console("Sensing for alerts.")();
            alert(msg);
            if(!is_defined(msg.self_req)){
                throw(alert)}
        }|{
            random@Math()(s_time);
            s_time=int(s_time*11);
```

```
33              println@Console("Waiting for alerts for: "+
34                  (s_time)+"s")();
35              for(i=s_time,i>0,i--){
36                  println@Console("-"+i)();
37                  sleep@Time(1000)()
38              };
39              throw(no_alert)
40          }};
41          println@Console("Alarm checked.")();
42          println@Console("System reset.")()
43      }
44  }
```

3.2.6.4 Critical Section

Two or more connected sub-graphs of a process model are identified as "critical sections".

At runtime for a given process instance, only activities in one of these "critical sections" can be active at any given time. Once execution of the activities in one "critical section" commences, it must complete before another "critical section" can commence.

Diagram

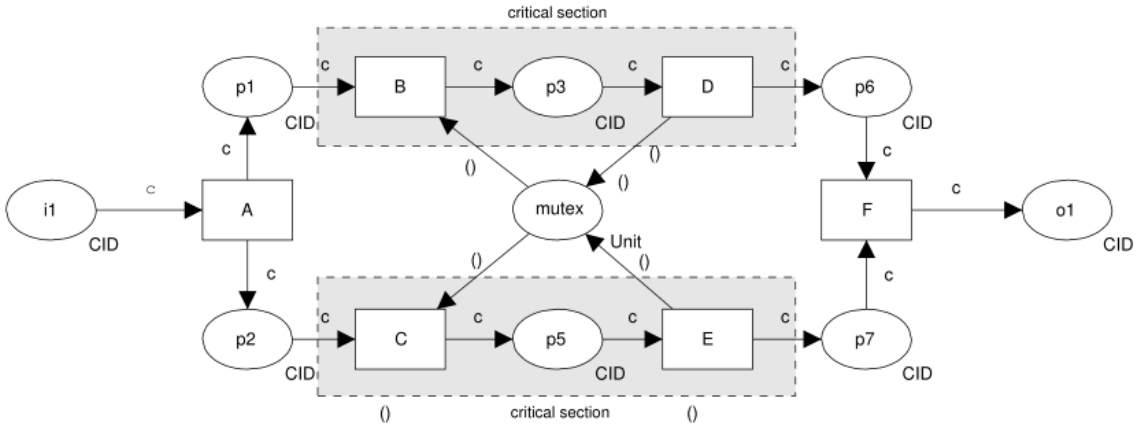


Figure 3.32: Critical Section pattern

Motivation

The *Critical Section* pattern provides a means of limiting two or more sections of a process from executing concurrently. Generally this is necessary if activities within this section require exclusive access to a common resource (either data or a physical resource) necessary for an activity to be completed.

However, there are also regulatory situations (e.g. as part of due diligence or quality assurance processes) which necessitate that two activities do not occur simultaneously.

JOLIE Implementation

There is one consideration associated with the use of this pattern: tasks must be initiated and completed on a sequential basis, in particular it is not possible to suspend one task during its execution to work on another.

Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption. Where an offering is able to achieve similar functionality through additional configuration or programmatic extension of its existing constructs (but does not have a specific construct for the pattern) this qualifies as partial support.

JOLIE implements the built-in mutually exclusive statement `synchronized(var)` which provides a synchronization primitive for programming mutual exclusion behaviors among concurrent sessions. More specifically, each branch's *Critical Section* is scoped within a synchronized globally shared token (lock) and all of its statements are executed atomically until it leaves the synchronization scope.

3.2.6.5 Interleaved Routing

Each member of a set of activities must be executed once. They can be executed in any order but no two activities can be executed at the same time (i.e. no two activities can be active for the same process instance at the same time).

Once all of the activities have completed, the next activity in the process can be initiated.

Diagram

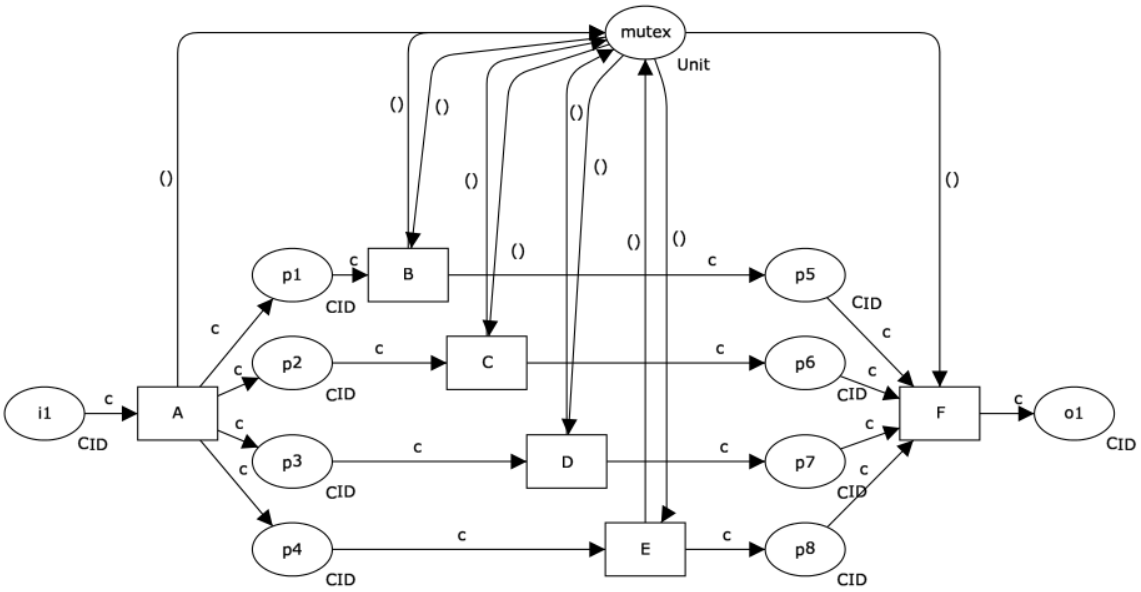


Figure 3.33: Interleaved Routing pattern

Motivation

The *Interleaved Routing* pattern relaxes the partial ordering constraint that exists with the *Interleaved Parallel Routing* pattern [3.2.6.2] and allows a sequence of activities to be executed in any order.

JOLIE **Implementation**

There are two considerations associated with the use of this pattern:

1. for a given process instance, it is not possible for two activities from the set of activities subject to interleaved routing to be executed at the same time;

2. activities must be initiated and completed on a sequential basis, in particular it is not possible to suspend one activity during its execution to work on another.

An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. An offering is rated as having partial support if it has limitations on the range of activities that can be coordinated (e.g. activities must be in the same process block) or if it cannot enforce that activities are executed precisely once or ensure activities are not able to be suspended once started whilst other activities in the interleave set are commenced.

Likewise for the *Critical Section* pattern [3.2.6.4], the built-in mutually exclusive statement `synchronized` can be employed to design mutual exclusion behaviors among concurrent sessions. Although being a relaxation of the *Interleaved Parallel Routing* patter, an approach towards an *Interleaved Routing* pattern implementation in JOLIE is fundamentally identical to the one described for the *Critical Section* pattern, in which each parallel branch is defined as a *Critical Section* by itself and executed sequentially (but not deterministically) w.r.t. all of the other by means of the same shared synchronization scope. The `spawn` construct is used to launch several parallel branches that share the same instruction sequence, but the same result can be achieved by declaring the same number of blocks in parallel composition.

JOLIE **code example**

Listing 51: Interleaved routing (server) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

main{
        exec_sequence->global.exec_sequence;
        spawn(i over 10) in arr{
        synchronized(lock){
            println@Console("Starting branch n."+(i+1))();
            sleep@Time(250)();
            exec_sequence=exec_sequence+(i+1)+"|"
        }
    };
```

```
14      println@Console("All branches executed."+
15          " Execution sequence was: "+exec_sequence)()
16  }
```

3.2.7 Cancellation and Force Completion Patterns

Several of the patterns above like the *Structured Synchronizing Merge* [3.2.4.2] and *Structured Discriminator* [3.2.4.4] have variants that utilize the concept of activity cancellation where enabled or active activity instance are withdrawn.

Various forms of exception handling in processes are also based on cancellation concepts.

It's noteworthy that the practice of canceling running tasks is a common and well known way to mess up processing data, which regularly ends up in data inconsistency and unpredictable process execution, if not directly managed by compensation policies.

Even if not directly linked to the *Cancellation and Force Completion Patterns* is remarkable that JOLIE offers a wide set of instructions and constructs that are specifically developed for fault handling, process termination, dynamic recovery and compensation.

3.2.7.1 Cancel Task

An enabled activity is withdrawn prior to it commencing execution. If the activity has started, it is disabled and, where possible, the currently running instance is halted and removed.

Diagram

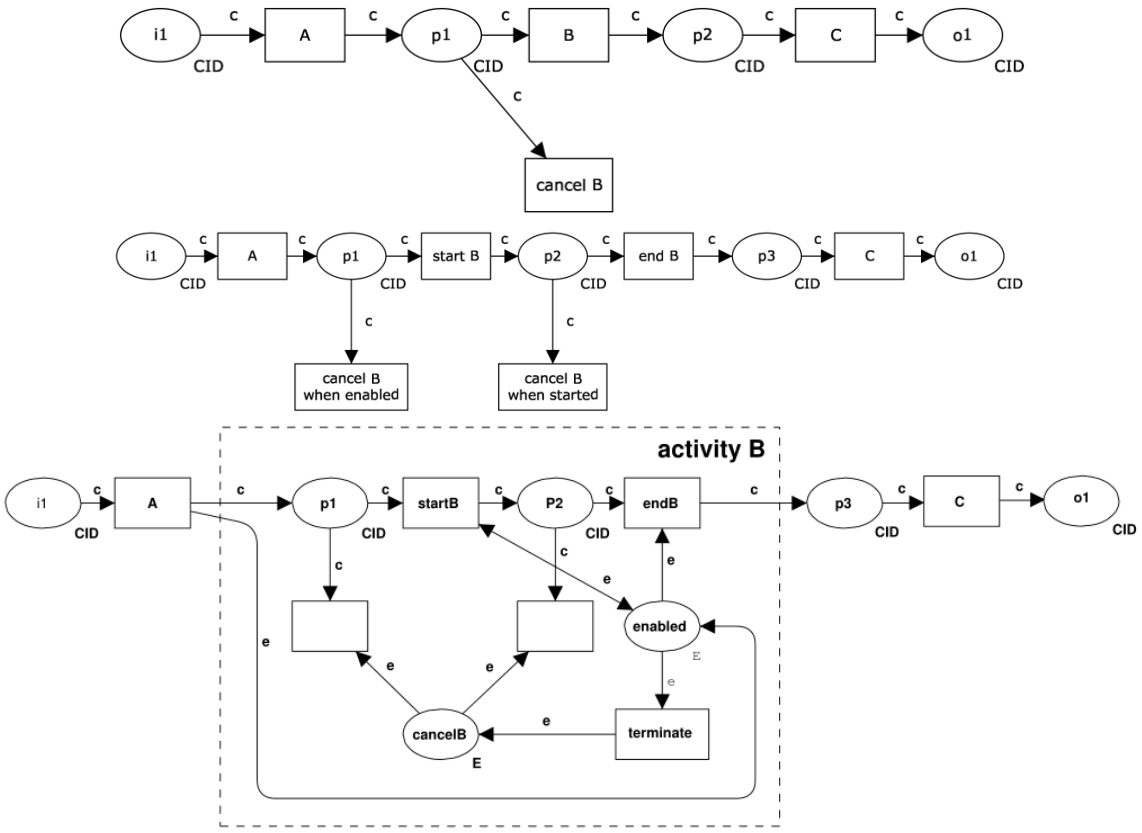


Figure 3.34: Cancel Task pattern (variants)

Motivation

The *Cancel Task* pattern provides the ability to withdraw an activity which has been enabled. This ensures that it will not commence execution.

JOLIE **Implementation**

There are 3 different interpretations of this pattern according to various contexts of execution (as illustrated in the previous pattern figures):

- the general interpretation of the *Cancel Task* pattern is the removal/disablement of the task's trigger, which prevents it from proceeding. This is done before the task's execution (first figure);

- the second variant of the pattern takes into account the eventuality that the task has already commenced its execution but has not yet completed. In this context a dedicated instruction is used to cancel the task;
- the latter variant is the guaranteed cancellation. It's worth noting that the previously mentioned interpretations of the *Canceling Task* do not take into account the necessity to guarantee the task's cancellation, since each of them simply defines a race between cancellation and execution, which can succeed or not according to threads execution. For this reason the last possible interpretation of the *Cancel Task* pattern makes allowance of the guarantee on the cancellation of the task. In such a context the decision to cancel a task can only be made after it has been enabled and prior to it completing. Once this decision is made, it is not possible for the activity to progress any further. For obvious reasons, it is not possible to cancel an activity which has not been enabled (i.e. there is no "memory" associated with the action of canceling an activity in the way that there is for triggers) nor is it possible to cancel an activity which has already completed execution.

An offering achieves full support for the pattern if it provides the ability to denote activity cancellation within a process model. If there are any side-effects associated with the cancellation (e.g. forced completion of other activities, the canceled activity being marked as complete), then the offering is rated as having partial support.

In JOLIE each of the possible interpretations are implementable:

- the first case is merely a *Deferred Choice* that is triggered before the task's execution, for this purpose it's made reference to the pattern [3.2.6.1];
- the second case can be obtained by means of a parallel process, scoped within the task code execution, which can be triggered by an external source and that rises an exception fault that stops the "sibling" process (the task's one) execution;
- the latter, the guaranteed cancellation case, can be achieved as a "combination" of a post-execution *Deferred Choice* and the solution described for the previous (second) variant of the defined pattern cases. Thus until the start of the process execution, no canceling action can take place, then the task

can be both canceled (therefore not marked as completed) during it's execution or at it's end, where a *Deferred Choice* is set to check if a terminating instruction has been fired during the process execution.

Since the trivial trait of the first interpretation and that the guaranteed cancellation variant includes in its behavior the second case, as it follows is provided the JOLIE code example for the third (guaranteed cancellation) case only.

JOLIE **code example**

Listing 52: Cancel Task (guaranteed cancellation variant) code example

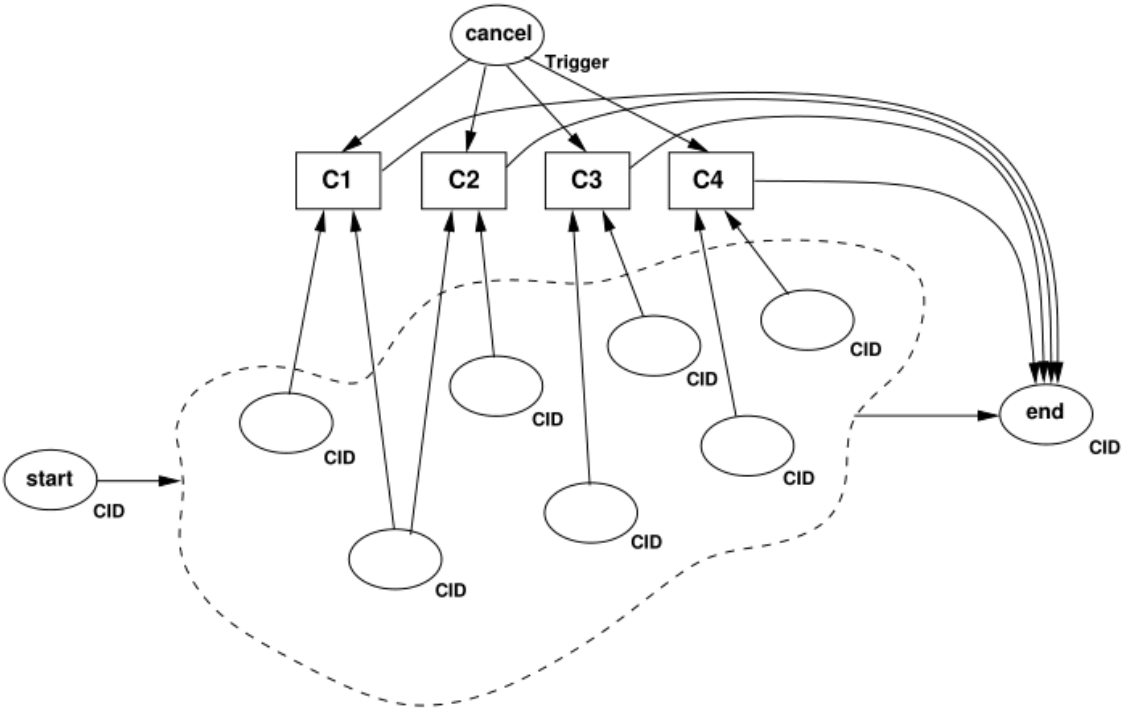
```
include "console.iol"
include "time.iol"
include "math.iol"

main{
    {
        install(terminate_process=>
            println@Console("Task cancelled")());
        println@Console("Task started")();
        linkOut(task_started);
        sleep@Time(3000)();
        synchronized(lock){
            if(!termination_token){
                println@Console("Task completed")()}
            else{
                println@Console("Termination"+
                    "signal received after completion. "+
                    "Canceling...")();
                throw(terminate_process)}
        }
    }|{
        linkIn(task_started);
        random@Math()(termination_sleep);
        sleep@Time(int(termination_sleep*4000))();
        synchronized(lock){termination_token=true};
        println@Console("Task termination triggered")();
        throw(terminate_process)
    }
}
```

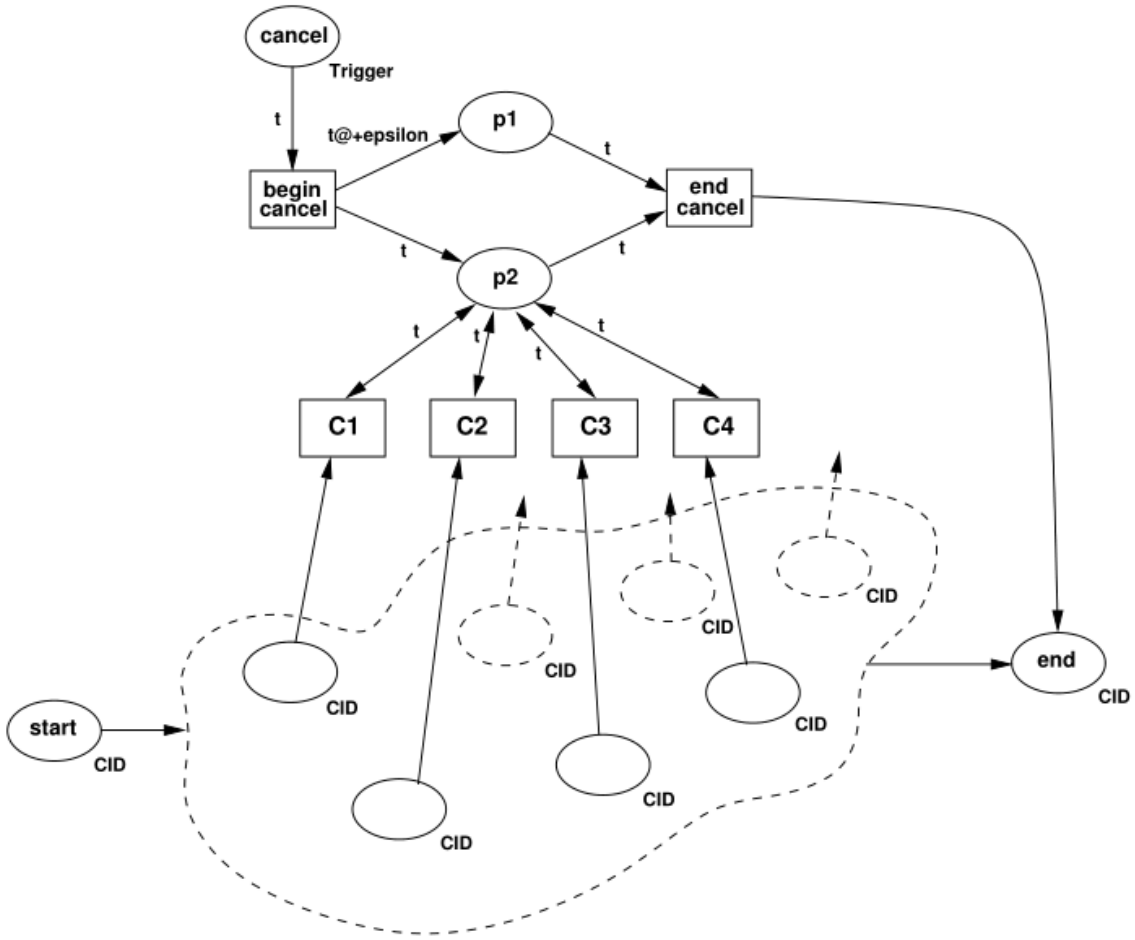
3.2.7.2 Cancel Case

A complete process instance is removed. This includes currently executing activities, those which may execute at some future time and all sub-processes.

The process instance is recorded as having completed unsuccessfully.

Diagram




cancel
Trigger
t
t@+epsilon
p1
t
begin cancel
end cancel
t
t
p2
t
t
t
t
C1
C2
C3
C4
CID
CID
CID
CID
CID
start
CID
CID
CID
CID
end
CID

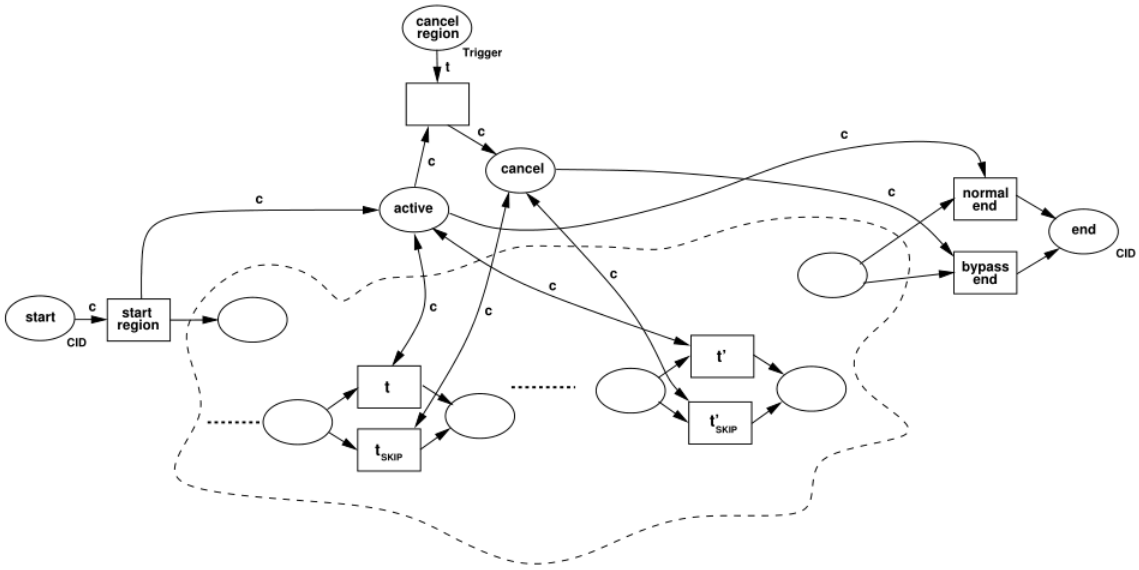


Figure 3.35: Cancel Case pattern (variants)

Motivation

The *Cancel Case* pattern provides a means of halting a specified process instance and withdrawing any activities associated with it.

JOLIE **Implementation**

As for the *Cancel Task* pattern [3.2.7.1], also for this pattern exist 3 different interpretations according to various contexts of execution (as illustrated in the previous pattern figures):

- the general interpretation of the *Cancel Case* pattern is the cancellation of an entire case, which involves the disabling of all currently enabled activities. Each combination has a transition associated with it that disables all enabled activities. Where cancellation of a case is enabled, it is assumed that precisely one of the canceling transitions will fire canceling all necessary enabled activities. To achieve this, it is necessary that none of the canceling transitions represent a state that is a super-set of another possible state, otherwise tokens may be left behind after the cancellation (first figure);

- the second variant of the pattern takes into account the case that every state has a set of cancellation transitions associated with it. When the cancellation is initiated, these transitions are enabled for a very short time interval, thus effecting an instantaneous cancellation for a given state that avoids the potential deadlocks that might arise with this approach.
- the third case is a more general approach to cancellation, in this case the pattern can be used to cancel individual activities, regions or even whole cases. It is premised on the creation of an alternative "bypass" activity for each activity in a process that may need to be canceled. When a cancellation is initiated, the case continues processing but the "bypass" activities are executed rather than the normal activities, so in effect no further work is actually achieved on the case.

There is an important context condition associated with this pattern: cancellation of a executing case must be viewed as unsuccessful completion of the case.

This means that even though the case was terminated in an orderly manner, perhaps even with tokens reaching its final end state, this should not be interpreted in any way as a successful outcome. For example, where a log is kept of events occurring during process execution, the case should be recorded as incomplete or canceled.

An offering achieves full support for the pattern if it provides the ability to denote the cancellation of an entire process instance in a process model and satisfies the context requirements for the pattern. If there are any side-effects associated with the cancellation (e.g. forced completion of other activities, the process instance being marked as complete), then the offering is rated as having partial support.

In JOLIE the first two variants of the pattern are implementable by means of the `install` and `throw` statements:

- The first case is merely a scoped execution (it's noteworthy that even the "main" statement is a scope by itself) of a process, which `installs` a fault handler for that instance and once the corresponding exception has been `thrown`, the execution is stopped and the process is marked as canceled;
- the same approach can be used for the second case, but in this context a finer grained control is available, since every state can be associated with both

static and dynamic `install` statements. Cumulative or separate termination behaviors can be implemented according to the context of execution, along with dedicated termination policies.

Finally the third case can be simply obtained by means of a sequence of *Deferred/Exclusive Choice* [3.2.3.4] blocks, one for each state transition block, which implements the "bypass" no-op mechanism described by the variant.

Since the first and second variant implementations of this pattern are very similar to the one defined for the guaranteed cancellation of the *Cancel Task* patter, is made reference to it for their implementation. The JOLIE code example of the third variant of the pattern is listed as it follows.

JOLIE code example

Listing 53: Cancel Case ("bypass" activity variant) code example

```
include "console.iol"
include "time.iol"

define state_transit{
    if(cancel_case){
        println@Console("NoOp.")()
    }else{
        println@Console("State transit."+i)();
        sleep@Time(1000)()
    }
}

main{
    {
        println@Console("Case started")();
        for(i=0,i<5,i++){
            state_transit
        };
        if(cancel_case){
        println@Console("Case Cancelled.")()
    }else{
        println@Console("Case Finished.")()
    }

    }|{
        sleep@Time(3000)();
        cancel_case=true
    }
}
```

3.2.7.3 Cancel Region

The ability to disable a set of activities in a process instance. If any of the activities are already executing, then they are withdrawn. The activities need not be a connected subset of the overall process model.

Diagram

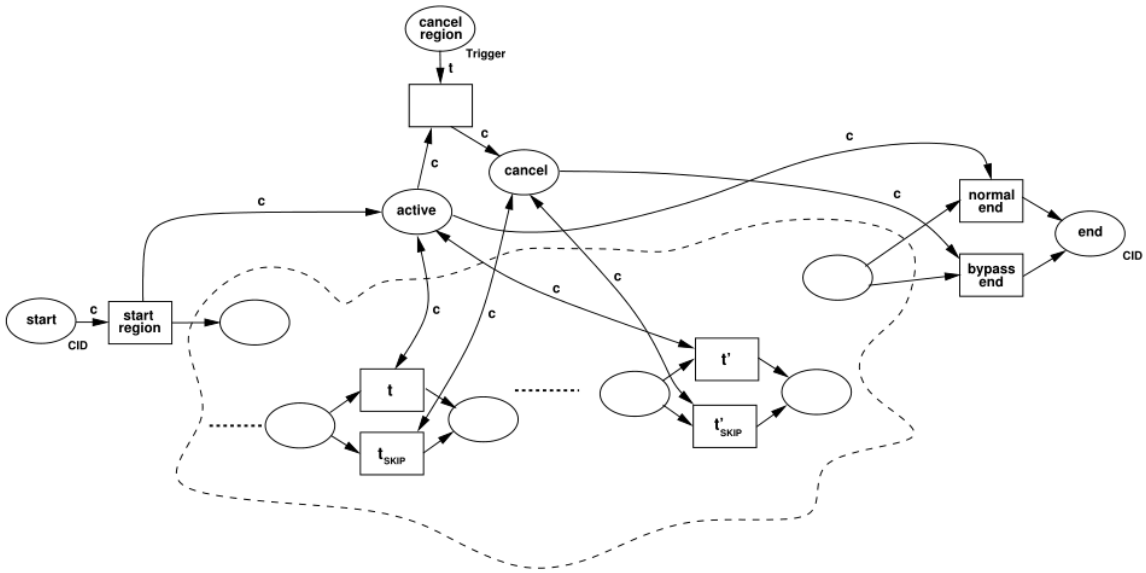


Figure 3.36: Cancel Region pattern

Motivation

The option of being able to cancel a series of (potentially unrelated) activities is a useful capability, particularly for handling unexpected errors or for implementing forms of exception handling.

JOLIE **Implementation**

The general form of this pattern is based on the premise that every activity in the required region has an alternate "bypass" activity. When the cancellation of the

region is required, the process instance continues execution, but the bypass activities are executed instead of the original activities. As a consequence, no further work occurs on the activities in the cancellation region. However, as shown for the *Cancel Case* pattern [3.2.7.2], there are several alternative mechanisms that can be used to cancel parts of a process.

There are two specific requirements for this pattern:

- it must be possible to denote a set of (not necessarily connected) activities that are to be canceled;
- once cancellation of the region is invoked, all activity instances within the region (both currently executing and also those that may execute at some future time) must be withdrawn.

One issue that can arise with the implementation of the *Cancel Region* pattern occurs when the canceling activity lies within the cancellation region. Although this activity must run to completion and cause the cancellation of all of the activities in the defined cancellation region, once this has been completed, it too must be canceled.

The most effective solution to this problem is to ensure that the canceling activity is the last of those to be processed (i.e. the last to be terminated) of the activities in the cancellation region.

An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern.

Since it's similarity to the *Cancel Case* "bypass" activity (third) variant, it's made reference to that JOLIE pattern for the *Cancel Region* pattern implementation.

3.2.7.4 Cancel Multiple Instance Activity

Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently.

At any time, the multiple instance activity can be canceled and any instances which have not completed are withdrawn. This does not affect activity instances that have already completed.

Diagram

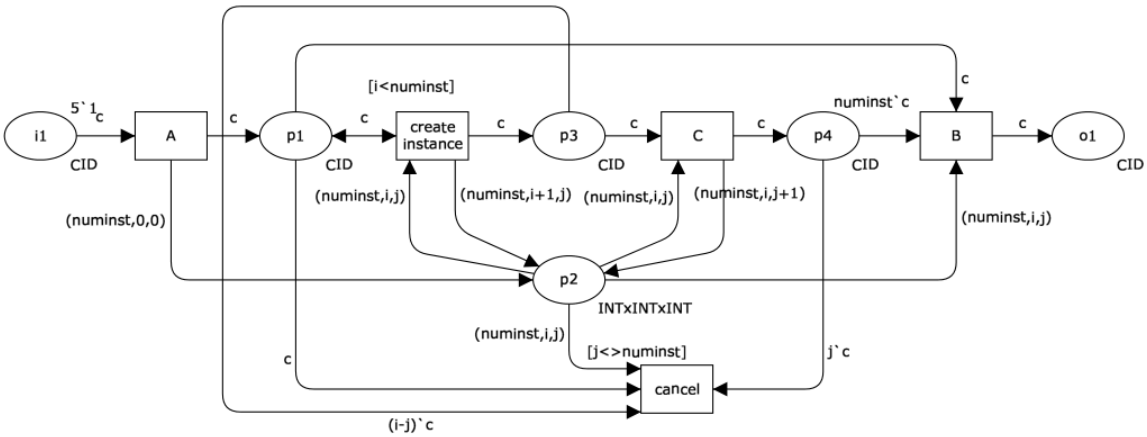


Figure 3.37: Cancel Multiple Instance Activity pattern

Motivation

This pattern provides a means of canceling a multiple instance activity at any time during its execution such that any remaining instances are canceled. However any instances which have already completed are unaffected by the cancellation.

JOLIE **Implementation**

There are two variants of this pattern depending on whether the activity instances are started sequentially or simultaneously. In both cases when the cancel transition is enabled, any remaining instances that have not already executed are withdrawn, as is the ability to initiate any additional instances. No subsequent activities are enabled as a consequence of the cancellation.

An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. If there are any limitations on the range of activities that can appear within the cancellation region or the types of activity instances that can be canceled then an offering achieves a partial rating.

Even considering the multiple instance context of this pattern, the same behavior described for the *Cancel Case* pattern [3.2.7.2] can be applied.

The execution of multiple instances can be synchronized by means of global variables and csets (concurrent execution), which make feasible implementing any composition of methods and contexts described previously: both the "bypass" or the `install-throw` variants can be easily implemented as long as both sequential and concurrent execution.

It's worth noting that a coarser grained approach, through a general block for the whole case, or a finer grained control via a state-by-state `install-throw` definition can be also adopted, resulting in a highly customizable, flexible and even mixable set of solutions made possible by the JOLIE .

Since the wide and varied feature of this pattern, which generates a significant number of alternative implementations according to the context and the required degree of control, it's made reference to a non obvious but interesting implementation of this pattern: the *Multiple Instances without a priori Run-Time Knowledge* pattern [3.2.5.4] code example, in which, after a defined event, each running instance is canceled and no other new instance can be invoke, while, as described by the *Cancel Multiple Instance Activity* pattern, all completed activities are unaffected by the cancellation.

3.2.7.5 Complete Multiple Instance Activity

Within a given process instance, multiple instances of an activity can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently.

It is necessary to synchronize the instances at completion before any subsequent activities can be triggered. During the course of execution, it is possible that the activity needs to be forcibly completed such that any remaining instances are withdrawn and the thread of control is passed to subsequent activities.

Diagram

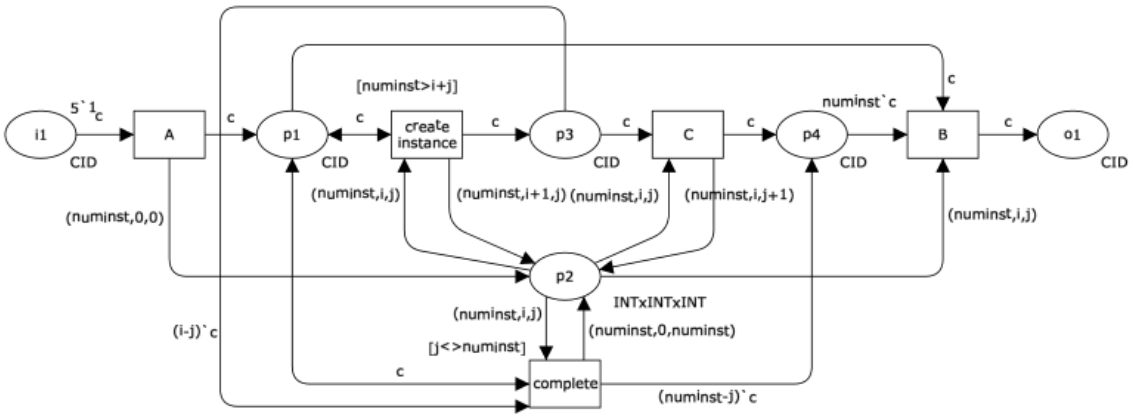


Figure 3.38: Complete Multiple Instance Activity pattern

Motivation

This pattern provides a means of finalizing a multiple instance activity that has not yet completed at any time during its execution such that any remaining instances are withdrawn and the thread of control is immediately passed to subsequent activities. Any instances which have already completed are unaffected by the cancellation.

JOLIE **Implementation**

There are two variants of this pattern depending on whether the task instances are started sequentially or simultaneously. In both cases when the complete transition is enabled, any remaining instances that have not already executed are withdrawn, as is the ability to add any additional instances. The subsequent task is enabled immediately.

There is one context condition associated with this pattern: only one instance of a multiple instance task can execute at any time.

An offering achieves full support if it provides a construct that satisfies the context requirements for the pattern. It demonstrates partial support if there are limitations on when the completion activity can be initiated or if the force completion of the remaining instances does not result in subsequent activities in the process instance being triggered normally.

Since it's resemblance to the *Canceling Partial Join for Multiple Instances* pattern [3.2.5.6], it's made reference to the *Canceling Partial Join* pattern [3.2.4.9] code example for the JOLIE implementation of this pattern, while applying the same concurrency considerations taken into account by the Multiple Instance variant of the pattern cited above.

3.2.8 Iteration Patterns

3.2.8.1 Arbitrary Cycles

The ability to represent cycles in a process model that have more than one entry or exit point.

Diagram

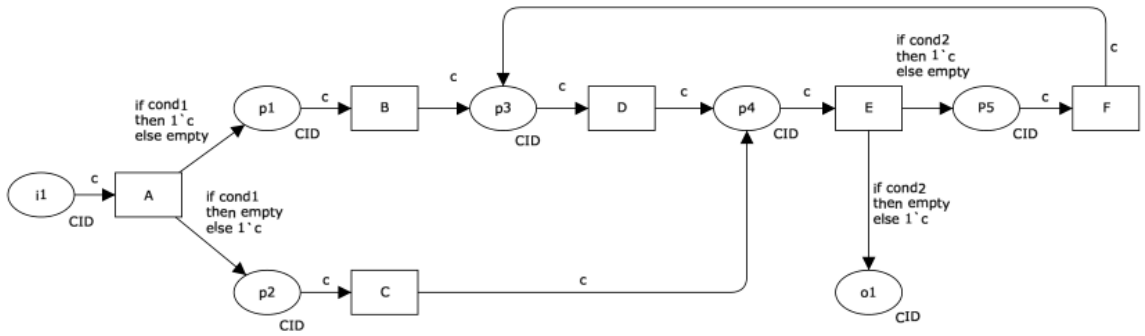


Figure 3.39: Arbitrary Cycles pattern

Motivation

The *Arbitrary Cycles* pattern provides a means of supporting repetition in a process model in an unstructured way without the need for specific looping operators or restrictions on the overall format of the process model.

JOLIE **Implementation**

There are no specific context conditions associated with the inclusion of arbitrary cycles in a process model other than the obvious requirement that the process model is able to support cycles (i.e. it is not block structured).

An offering achieves full support for the pattern if it is able to capture unstructured cycles that have more than one entry or exit point.

At the moment the JOLIE does not offer any support to this pattern.

3.2.8.2 Structured Loop

The ability to execute an activity or sub-process repeatedly. The loop has either a pre-test or post-test condition associated with it that is either evaluated at the beginning or end of the loop to determine whether it should continue. The looping structure has a single entry and exit point.

Diagram

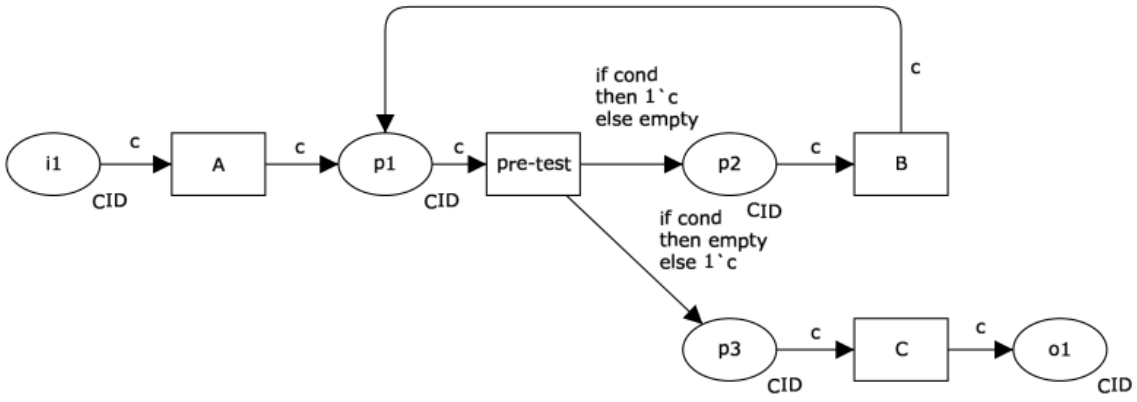


Figure 3.40: Structured Loop pattern

Motivation

There are two general forms of this pattern:

- the *while* loop which equates to the classic `while...do` pre-test loop construct used in programming languages;
- the *repeat* loop which equates to the `repeat...until` post-test loop construct.

The while loop allows for the repeated sequential execution of a specified activity or a sub-process zero or more times providing a nominated condition evaluates to true. The pre-test condition is evaluated before the first iteration of the loop and is

re-evaluated before each subsequent iteration. Once the pre-test condition evaluates to false, the thread of control passes to the activity immediately following the loop. The while loop structure ensures that each of the activities embodied within it are executed the same number of times.

The repeat loop allows for the execution of an activity or sub-process one or more times, continuing with execution until a nominated condition evaluates to true. The post-test condition is evaluated after the first iteration of the loop and is re-evaluated after each subsequent iteration. Once the post-test condition evaluates to true, the thread of control passes to the activity immediately following the loop. The repeat loop structure ensures that each of the activities embodied within it are executed the same number of times.

JOLIE **Implementation**

An offering achieves full support for the pattern if it has a construct that denotes an activity or sub-process should be repeated whilst a specified condition remains true or until a specified condition becomes true.

JOLIE offers a set of primitives for the *Structured Loop* pattern implementation like the `for` and `foreach` (used for navigating structures) constructs which implements the *repeat* loop form of the pattern, along with the `while` construct.

3.2.8.3 Recursion

The ability of an activity to invoke itself during its execution or an ancestor in terms of the overall decomposition structure with which it is associated.

Diagram

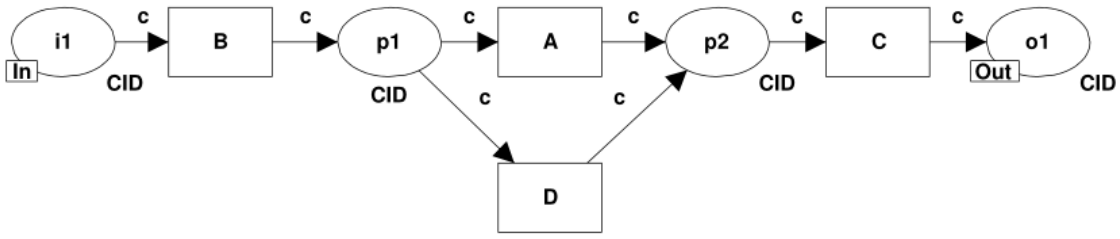


Figure 3.41: Recursion pattern

Motivation

For some types of activity, particularly those that may involve unplanned repetition of an activity or sub-process, simpler and more succinct solutions can be provided through the use of recursion rather than iteration. In order to harness recursive forms of problem solving within the context of a workflow, a means of describing an activity execution in terms of itself (i.e. the ability for an activity to invoke another instance of itself whilst executing) are required.

JOLIE **Implementation**

An offering achieves full support if it it is able to satisfy the context criteria for the pattern.

JOLIE does not support natively the *Recursion* pattern, although a similar behavior is achieved by a self-OneWay request as reported in the code example as it follows.

JOLIE **code example**

Listing 54: Recursion (workaround) code example

```
include "console.iol"
include "time.iol"

outputPort REC{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: rec
}

inputPort REC{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: rec
}

execution{concurrent}

init{
    rec@REC(1)
}

main{
    rec(msg);
    println@Console("Message number: "+msg)();
    sleep@Time(1000)();
    msg++;
    rec@REC(msg)
}
```

3.2.9 Termination Patterns

3.2.9.1 Implicit Termination

A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future.

Motivation

The rationale for this pattern is that it represents the most realistic approach to determining when a process instance can be designated as complete. This is when there is no remaining work to be completed as part of it and it is not possible that work items will arise at some future time.

JOLIE **Implementation**

Where an offering does not directly support this pattern, the question arises as to whether it can implement a process model which has been developed based on the notion of implicit termination.

For simple process models, it may be possible to indirectly achieve the same effect by replacing all of the end nodes for a process with links to a *Simple Merge* pattern [3.2.3.5] which then links to a single final node.

However, it is less clear for more complex process models involving multiple instance activities whether they are always able to be converted to a model with a single terminating node. It is worthwhile noting that some languages do not offer this construct on purpose: the *Implicit Termination* pattern makes it difficult (or even impossible) to distinguish proper termination from deadlock! Additionally, workflows without explicit endpoints are more difficult to use in compositions.

An offering achieves full support for this pattern if process (or sub-process) instances terminate when there are no remaining activities to be completed now or at any time in the future and the process instance is not in deadlock.

JOLIE implements natively the kind of behavior described by this pattern.

3.2.9.2 Explicit Termination

A given process (or sub-process) instance should terminate when it reaches a nominated state.

Typically this is denoted by a specific end node. When this end node is reached, any remaining work in the process instance is canceled and the overall process instance is recorded as having completed successfully.

Motivation

The rationale for this pattern is that it represents an alternative means of defining when a process instance can be designated as complete. This is when the thread of control reaches a defined state within the process model. Typically this is denoted by a designated termination node at the end of the model.

JOLIE **Implementation**

There are two specific context conditions associated with this pattern:

- every activity in a the process must be on a path from a defined starting node to a defined end node;
- when the thread of control reaches the end node, the process is deemed to have completed successfully regardless of whether there are any activities in progress or remaining to be executed, for example, where a log is kept of process activity, the process instance would be recorded as completing successfully.

One consideration that does arise where a process model has multiple end nodes is whether it can be transformed to one with a single end node. For simple process models, it may be possible to simply replace all of the end nodes for a process with links to a *Simple Merge* pattern [3.2.3.5] in which then links to a single final node. However, it is less clear for more complex process models involving multiple instance activities whether they are always able to be converted to a model with a single terminating node.

An offering achieves full support for this pattern if it demonstrates that it can meet the context requirements for the pattern.

One of the possible JOLIE implementation examples that realizes the behavior described by this pattern, is the code example provided with the *Structured Discriminator* pattern [3.2.4.4] in which, after one of the branches has completed (and

has reached its “end node”) all other are discarded while the process is deemed as completed successfully.

3.2.10 Trigger Patterns

3.2.10.1 Transient Trigger

The ability for an activity to be triggered by a signal from another part of the process or from the external environment. These triggers are transient in nature and are lost if not acted on immediately by the receiving activity.

Diagram

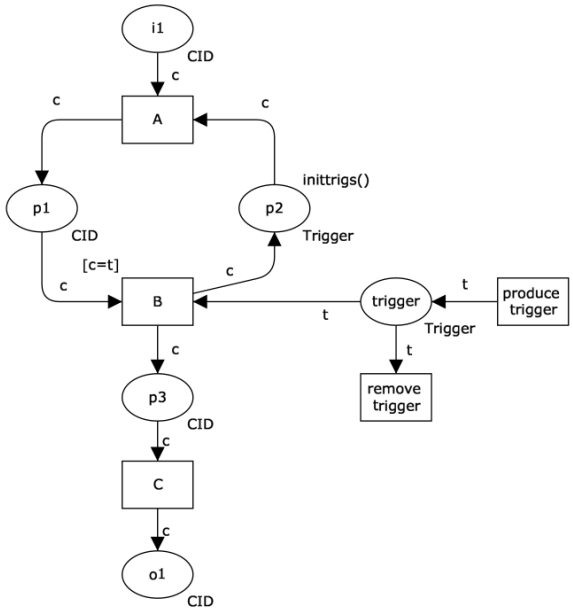


Figure 3.42: Transient Trigger (safe variant) pattern

Motivation

Transient triggers are a common means of signaling that a predefined event has occurred and that an appropriate handling response should be undertaken comprising either the initiation of a single activity, a sequence of activities or a new thread of execution in a process. Transient triggers are events which must be dealt

with as soon as they are received. In other words, they must result in the immediate initiation of an activity. The workflow provides no form of memory for transient triggers. If they are not acted on immediately, they are irrevocably lost.

JOLIE **Implementation**

Transient triggers have two context conditions associated with them:

- it must be possible to direct a trigger to a specific activity instance executing in a specific process instance;
- if the activity instance to which the trigger is directed is not waiting (for the trigger) at the time that the trigger is received, then the trigger is lost.

There are two main variants of this pattern depending on whether the process is executing in a safe execution environment or not. In the safe variant, only one instance of activity can wait on a trigger at any given time. The alternative option for unsafe processes let multiple instances remain waiting for a trigger to be received. However only one of these can be enabled for each trigger when it is received.

One consideration that arises with the use of transient triggers is what happens when multiple triggers are received simultaneously or in a very short time interval. Are the latter triggers inherently lost as a trigger instance is already pending or are all instances preserved (albeit for a potentially short timeframe).

In general, in the implementations examined it seems that all transient triggers are lost if they are not immediately consumed. There is no provision for transient triggers to be duplicated.

An offering achieves full support if it it is able to satisfy the context criteria for the pattern.

One of the possible implementations of this pattern in JOLIE, is based on a parallel process that can be invoked and which releases a token for a certain period of time, after which the token is discarded. Safe or unsafe variants are modeled by making use of the `synchronized` construct.

JOLIE **code example**

Listing 55: Transient Trigger (safe) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

main{
    {
        while(true){
            sleep@Time(3000)();
            println@Console("Triggered alert")();
            synchronized(lock){token.alert=true};
            sleep@Time(1000)();
            synchronized(lock){token.alert=false};
            println@Console("Alert over")()
        }
    }|{
        while(true){
            {
                sleep@Time(3000)();
                synchronized(lock){
                    if(token.alert){
                        exec1=true}};
                synchronized(exec){
                    if(exec1){
                        println@Console("Process 1 "+
                            "triggered")();
                        sleep@Time(3000)();
                        println@Console("Process 1 "+
                            "finished")()
                    }}
            }|{
                sleep@Time(3000)();
                synchronized(lock){
                    if(token.alert){
                        exec2=true}};
                synchronized(exec){
                    if(exec2){
                        println@Console("Process 2 "+
                            "triggered")();
                        sleep@Time(3000)();
                        println@Console("Process 2 "+
                            "finished")()
                    }}

            }|{
                sleep@Time(6000)();
                synchronized(lock){
                    if(token.alert){
                        exec3=true}};
```

```
49                  synchronized(exec){
50                      if(exec3){
51                          println@Console("Process 3 "+
52                              "triggered")();
53                          sleep@Time(3000)();
54                          println@Console("Process 3 "+
55                              "finished")()
56                  }}
57
58              }
59          }
60      }
61  }
```

3.2.10.2 Persistent Trigger

The ability for an activity to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the workflow until they can be acted on by the receiving activity.

Diagram

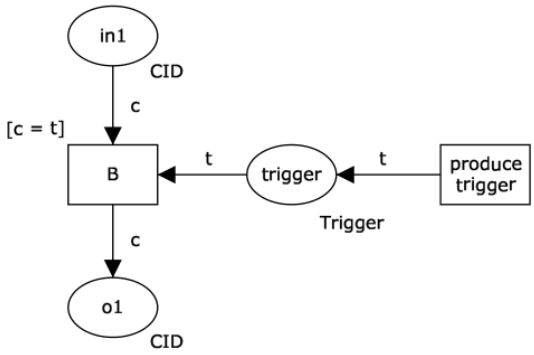


Figure 3.43: Persistent Trigger (control-flow variant) pattern

Motivation

Persistent triggers are inherently durable in nature, ensuring that they are not lost in transit and are buffered until they can be dealt with by the target activity. This means that the signaling activity can be certain that the trigger will result in the activity to which they are directed being initiated either immediately (if it already has received the thread of control) or at some future time.

JOLIE **Implementation**

There are two variants of the persistent triggers. One where a trigger is buffered until control-flow passes to the activity to which the trigger is directed, once this activity has received a trigger, it can commence execution.

Alternatively, the trigger can initiate an activity (or the beginning of a thread of execution) that is not contingent on the completion of any preceding activities.

An offering achieves full support for this pattern if it provides any form of durable activity triggering that can be initiated from outside the process environment. If

triggers do not retain a discrete identity when received and/or stored, an offering is viewed as providing partial support.

`LinkIn` and `linkOut` statements can represent a valid and native implementation of this pattern, even if a specific trigger must be defined for each activity.

An alternative implementation can be obtained by means of a parallel operation, executed along with the main process, which can be invoked and is able to manage each trigger's identity, by storing them into a shared token-based structure among the activities. This method eliminates the necessity to define a different trigger for every activity at design-time and preserves each trigger identity too.

Both the implementations defined above can realize either the control-flow and the not-contingent variants since their behavior is defined by the context the operation is working in.

As it follows it's reported the `LinkIn-linkOut` implementation of this pattern.

JOLIE **code example**

Listing 56: Persistent Trigger code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define trig1{
    random@Math()(sleep_time1);
    sleep_time1=int(sleep_time1*10000);
    sleep@Time(sleep_time1)();
    println@Console("Trigger 1 released after "+
        (sleep_time1/1000)+"s.")();
    linkOut(trig1)
}

define trig2{
    random@Math()(sleep_time2);
    sleep_time2=int(sleep_time2*10000);
    sleep@Time(sleep_time2)();
    println@Console("Trigger 2 released after "+
        (sleep_time2/1000)+"s.")();
    linkOut(trig2)
}

define trig3{
    random@Math()(sleep_time3);
    sleep_time3=int(sleep_time3*10000);
    sleep@Time(sleep_time3)();
    println@Console("Trigger 3 released after "+
```

```
28          (sleep_time3/1000)+"s.")();
29      linkOut(trig3)
30  }
31
32  main{
33      trig1|
34      trig2|
35      trig3|
36      scope(trig1){
37          install(timeout=>
38              println@Console("Process 1: Operation timed out.")());
39          install(triggered=>
40              println@Console("Process 1: Operation triggered.")());
41          {
42              sleep@Time(5000)();
43              throw(timeout)
44
45          }|{
46              linkIn(trig1);
47              throw(triggered)
48          }
49      }|
50      scope(trig2){
51          install(timeout=>
52              println@Console("Process 2: Operation timed out.")());
53          install(triggered=>
54              println@Console("Process 2: Operation triggered.")());
55          {
56              sleep@Time(5000)();
57              throw(timeout)
58
59          }|{
60              linkIn(trig2);
61              throw(triggered)
62          }
63      }|
64      scope(trig3){
65          install(timeout=>
66              println@Console("Process 3: Operation timed out.")());
67          install(triggered=>
68              println@Console("Process 3: Operation triggered.")());
69          {
70              sleep@Time(5000)();
71              throw(timeout)
72
73          }|{
74              linkIn(trig3);
75              throw(triggered)
76          }
77      }
78  }
```

3.3 Summary Table of JOLIE Control-Flow Patterns Support

Control-Flow Pattern	JOLIE **Support**
Sequence [3.2.3.1]	+
Parallel Split [3.2.3.2]	+
Synchronization [3.2.3.3]	+
Exclusive-Choice [3.2.3.4]	+
Simple-Merge [3.2.3.5]	+
Multi-Choice [3.2.4.1]	+
Structured Synchronization [3.2.4.2]	+
Multi-Merge [3.2.4.3]	+
Structured Discriminator [3.2.4.4]	+
Blocking Discriminator [3.2.4.5]	+
Canceling Discriminator [3.2.4.6]	+
Structured Partial Join [3.2.4.7]	+
Blocking Partial Join [3.2.4.8]	+
Canceling Partial Join [3.2.4.9]	+
General AND-Join [3.2.4.10]	+
Local Synchronization Merge [3.2.4.11]	+
General Synchronization Merge [3.2.4.12]	+
Thread Merge [3.2.4.13]	+
Thread Split [3.2.4.14]	+

Table 3.1: The table provided lists the first 19 Control-Flow patterns analyzed previously. Following the same convention adopted by WPI if a pattern can be realized in directly it is rated +, if is not directly supported, but has been realized through a workaround is rated +/-, finally if no implementation is supported is rated -.

Control-Flow Pattern	JOLIE **Support**
Multiple Instances without Synchronization [3.2.5.1]	+
Multiple Instances without a Priori Design-Time Knowledge [3.2.5.2]	+
Multiple Instances with a Priori Run-Time Knowledge [3.2.5.3]	+
Multiple Instances without a Priori Run-Time Knowledge [3.2.5.4]	+
Static Partial Join for Multiple Instances [3.2.5.5]	+
Canceling Partial Join for Multiple Instances [3.2.5.6]	+
Dynamic Partial Join for Multiple Instances [3.2.5.7]	+
Deferred Choice [3.2.6.1]	+
Interleaved Routing [3.2.6.2]	+
Milestone [3.2.6.3]	+
Critical Section [3.2.6.4]	+
Interleaved Routing [3.2.6.5]	+
Cancel Task [3.2.7.1]	+
Cancel Case [3.2.7.2]	+
Cancel Region [3.2.7.3]	+
Cancel Multiple Instance Activity [3.2.7.4]	+
Complete Multiple Instance Activity [3.2.7.5]	+
Arbitrary Cycles [3.2.8.1]	-
Structured Loop [3.2.8.2]	+
Recursion [3.2.8.3]	+/-
Implicit Termination [3.2.9.1]	+
Explicit Termination [3.2.9.2]	+
Transient Trigger [3.2.10.1]	+
Persistent Trigger [3.2.10.2]	+

Table 3.2: The table provided lists the last 24 Control-Flow patterns analyzed previously. Following the same convention adopted by WPI if a pattern can be realized in directly it is rated +, if is not directly supported, but has been realized through a workaround is rated +/-, finally if no implementation is supported is rated -.

3.4 Resource Patterns

3.4.1 What a "Resource" is

For a better understanding of the purposes of Resource Patterns, fundamental to clearly define what a "*Resource*" is.

Basically a resource can be considered as an entity that is capable of doing work. Thus, speaking about resources introduces a new term that's the *work item*, a work item can be describe as an integral unit of work that the resource should undertake.

A resource can be classified as either human or non-human with the difference that a human resource is typically a member of an organization; such an organization is a formal grouping of resources that undertake work items pertaining to a common set of business objectives. Human resources usually have a specific position within that organization and in general, most organizational characteristics belonging to a resource relate to the position(s) occupied by it, rather than directly to the resource itself.

As a consequence of their position(s), resources may have a number of associated privileges. They may also be a member of one or more organizational units, which are permanent groups of human resources within the organization that undertake work items relating to a common set of business objectives.

Similarly they may also be members of one or more organizational teams. These are similar to organizational units but not necessarily permanent in nature. Even less formal in nature, it's the notion of organizational groups which are often used to define groupings of resources with some common characteristic or cause e.g. social club members, fire-wardens etc. Each resource is generally associated with a specific branch, which defines a grouping of resources within the organization at a specific physical location.

Resources may also have a level which indicates their position within the organizational hierarchy. They may belong to a division too, which defines a large scale grouping of resources within an organization, either along regional geographic or business purpose lines. In terms of the organizational hierarchy, each resource may have a number of specific relationships with other resources.

Their direct report is the resource to whom they are responsible for their work,

generally these are more "senior" resource, at a higher organizational level. Similarly, a resource may also have a number of subordinates for whom they are responsible and to which each of them reports.

Finally, a resource may also have a delegate which is an alternate human resource to which they assign work items previously allocated to them. This reassignment of work items may occur on a temporary or permanent basis. A resource may have one or more associated roles. Roles serve as another grouping mechanism for human resources with similar job roles or responsibility levels e.g. managers, union delegates etc. Individual resources may also possess capabilities or attributes that further clarify their suitability for various kinds of work items. These may include qualifications and skills as well as other job-related or personal attributes such as specific responsibilities held or previous work experience. They may also have features which further describe specific characteristics that they may possess that could be of interest when allocating work items.

Non-human resources may be durable or consumable in nature. A durable resource is one whose capacity to undertake work is unaffected by the amount of work that it has undertaken, whereas a consumable resource is one that is consumed (either partially or wholly) in the act of completing a work item. There is usually a rate of consumption or capacity associated with consumable resources indicating how much work they can actually undertake before being depleted and requiring further replenishment. Each resource may have a schedule and history associated with them. These are essentially inverses of each other.

A *schedule* is a list of work items that a resource is committed to undertaking at a specified future times where as a *history* or *work log* is a list of work items that a resource has completed (successfully or otherwise) at some time in the past.

3.4.2 Adopted Conventions

3.4.2.1 Human resources, non-Human resources and patterns implementations in JOLIE

As stated in the preceding section, the definition of a "resource" can be declined into two categories: human and non-human resources, whose leading difference is that a human resource is defined as a member of an organization which bears a lot of characteristics with it like position(s), roles, groups (with corresponding

qualifications and skills), privileges, permanent or temporary basis, hierarchy and responsibilities.

Contrariwise, non-human resources are simply describe as durable or consumable ones that can undertake one or more work items, independently from any hierarchical, grouping or responsibility definition.

Although this definition is clearly correct with respect to the context defined before, in the circumstances of this work a human-like behavior is simulated by non-human resources i.e. automated processes. Even though such processes are non-human, their are structured, grouped and designed to simulate the behavior of a human resource with reference to the specific pattern situation.

This assumption is actually made since, as a matter of facts, a BPM language has a double nature of a modeling and deployment language. Such characteristic is exemplified by the fact that such a language can be both used to formally represent business models with a certain degree of abstraction and to deploy and run that model by means of a language-compliant execution engine.

As defined when writing about SOC, these model are based on connecting multiple resources, whose work on items shared among them is redirected, aggregated and elaborated by the orchestrator to achieve a certain task.

Thus, even if the strict definition of human and non-human resource states that only human ones can be considered part of an organization, in this work and for its replication purposes, human resources are represented by non-human ones that simulate their behaviors by means of organizational structuration, role assignation and the like.

3.4.3 Resource Patterns and Workflow Structures

As stated for the description of Control-Flow patterns, also Resource Patterns need a specific representational language which can be used to sufficiently detail the pattern depiction.

In its work in modeling Resource Patterns the WPI has chosen a *Workflow Model* (WM) which is the resulting composition of a number of *tasks* connected of the form of a directed graph.

In the workflow model a *process instance* is defined as a *case* and multiple cases of a particular WM can run simultaneously. It's noteworthy that in all of considered

pattern contexts each case is assumed as independent and executed without any reference to each other.

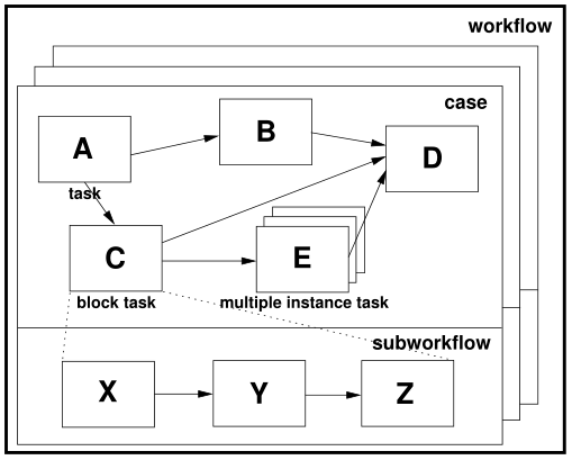


Figure 3.44: Workflow Model components

The Figure 3.44 depicts the various components of a WM. In each WM there are usually two particular unique tasks: the first and the last to run in the workflow case.

A *task* corresponds to a single unit of work, but four distinct types of tasks can be denoted:

- *atomic*: it's a task which has a simple and self-contained definition (i.e. not describable in terms of other workflow tasks) and only one instance of the task can be executed when initiated. W.r.t. Figure 3.44, *atomic* tasks are *A*,*B*,*X*,*Y*,*Z* and *D*.

- *block*: it represents a complex action which has its implementation describe in terms of a *sub-workflow* (SW). Thus when started, the *block task* passes its control to the first task in its corresponding SW. Once executed to completion (last task), the control is passed back to the block task. W.r.t. Figure 3.44 *C* is a block task.

- *multi-instance*: it's a task which may have many distinct execution instances running concurrently within the same workflow case. Each of these instances are executed independently and only once a nominated number of

these have completed, the task following the *multi-instance* task can be initiated. W.r.t. Figure 3.44 *E* is a *multi-instance* task.

- *multiple-instance block*: it's the resulting combination of a *multi-instance* and a *block task* which has multiple distinct execution instances each of which composed according to a *block task* structure.

The solid arrows between tasks indicate control flow passages among them, namely a *control channel* (i.e. each solid arrow in Figure 3.44 is a control channel). Each invocation of a task in called *work item* and, unless defined differently (i.e. *multiple-instance* tasks), it's assumed that each task, in a given case, has only one work item initiated. It's noteworthy that in loop cycles, each iteration creates a distinct work item for each task composing the loop.

3.4.3.1 Work distribution to resources

Once defined the concepts of tasks and work items in the context of resource patterns, it's important to describe how work items are advertised and bound to a specific *resource* of execution.

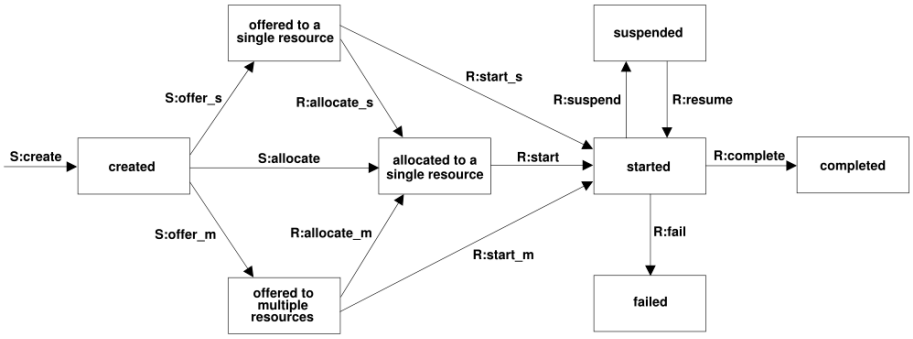


Figure 3.45: Work Item Lifecycle

Figure 3.45 illustrates a work item lifecycle from its creation to its completion or failure. As the cited figure depicts, a work item come into existence in the *created* state. It's worth noting that the incoming edge (channel control) of this state indicates an *S:create* precondition in which the prefix *S* identifies the initiator of the transition - *S* for workflow system, *R* for resource.

Once the workflow system has created a work item, it can inform exactly one resource about the availability of a work item (by means of a message to the resource or adding the work item into the list of those available to that resource). Alternatively, the system can inform a multitude of resources about the availability of a work item which will compete for the work item acquisition.

In the following section work item creation patterns will be taken into account and defined properly.

3.4.4 Creation Patterns

Creation Patterns correspond to limitations on the manner in which a work item may be executed. They are specified at design time, usually in relation to a task, and serve to restrict the range of resources that can undertake work items corresponding to the task. They also influence the manner in which a work item can be matched with a resource that is capable of undertaking it.

The essential rationale for creation patterns is that they provide a degree of clarity about how a work item should be handled after creation during the offering and allocation stages prior to it being executed. This ensures that the operation of a process conforms with its intended design principles and operates as efficiently and deterministically as possible.

In terms of the work item life-cycle, creation patterns come into effect at the time a work item is created. This state transition occurs at the beginning of the work item lifetime and it's depicted in Figure 3.45.

For all of these patterns it is assumed that there is an associated organizational model which allows resources to be uniquely identified and that there is a mechanism to distribute work items to specific resources identified in the organizational model. As creation patterns are specified at design time, they usually form part of the process model which describes a business process.

3.4.4.1 Direct Distribution

The ability to specify at design time the identity of the resource(s) to which instances of this task will be distributed at runtime.

Motivation

Direct allocation offers the ability for a workflow designer to precisely specify the identity of the resource to which instances of each task will be allocated at runtime.

This is particularly useful where it is known that a task can only be effectively undertaken by a specific resource as it prevents the problem of unexpected or non-suitable resource allocations arising at runtime by ensuring work items are routed to specific resources, a feature that is particularly desirable for critical tasks.

JOLIE Implementation

In JOLIE single work items can be directly distributed to their corresponding tasks by means of their identifier (name).

3.4.4.2 Role-Based Distribution

The ability to specify at design-time one or more roles to which instances of this task will be distributed at runtime.

Roles serve as a means of grouping resources with similar characteristics. Where an instance of a task is distributed in this way, it is distributed to all resources that are members of the role(s) associated with the task.

Motivation

Role-Based Distribution is the most common approach to work item allocation within workflow systems, role-based allocation offers the means for the workflow engine to route work items to suitably qualified resources at run-time. The decision as to which resource actually receives a given work item is deferred until the moment at which it becomes "runnable" and requires a resource allocation in order for it to proceed.

The advantage offered by role-based allocation (over other work item allocation schemes) is that roles can be defined for a given workflow process that define the various classes of available resources to undertake work items. Task definitions within the process model can nominate the specific role to which they should be routed, however the actual population of individual roles does not need to occur until run-time.

JOLIE **Implementation**

An offering achieves full support if it satisfies the description for the pattern.

In JOLIE work items can be distributed based on role by means of data structures which can contain a pool of items which are visible by tasks with the same role.

JOLIE **code example**

Listing 57: Role-Based Distribution code example

```
include "console.iol"

define pop_alert{
    if(#msg.alert>1){
        for(i=0,i<#msg.alert,i++){
        msg.alert[i]=msg.alert[i+1]
    }};
    undef(msg.alert[#msg.alert-1])
}
```

```
10
11 define pop_warning{
12     if(#msg.warning>1){
13         for(i=0,i<#msg.warning,i++){
14         msg.warning[i]=msg.warning[i+1]
15     }};
16     undef(msg.warning[#msg.warning-1])
17 }
18
19 main{
20     msg.alert[0]="This is alert 1";
21     msg.alert[1]="This is alert 2";
22     msg.warning[0]="This is warning 1";
23     {{
24         synchronized(alert){
25             println@Console("Alert msg: "+msg.alert[0])();
26             pop_alert}
27     }|{
28         println@Console("Warning msg: "+msg.warning[0])();
29         pop_warning
30     }|{
31         synchronized(alert){
32             println@Console("Alert msg: "+msg.alert[0])();
33             pop_alert}
34     }}
35 }
```

3.4.4.3 Deferred Distribution

The ability to specify at design-time that the identification of the resource(s) to which instances of this task will be distributed will be deferred until runtime.

Motivation

Deferred Distribution takes the notion of indirect work distribution one step further and allows the process designer to defer the need to identify the resource for a specific task (or work items corresponding to the task) until runtime.

One means of achieving this is to nominate a data field from which the identity of the resource to which a work item should be routed can be determined at runtime. The identity of the resource can be changed dynamically during process execution by updating the value of the data field, thus varying the resource allocation of future work items which are contingent on it.

JOLIE **Implementation**

There is one context condition associated with this pattern: the offering supports direct or role-based distribution.

Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption.

A JOLIE implementation of the *Deferred Distribution* pattern can be obtained as a slight modification of the *Role-Based Distribution* pattern code example in which alert and warning messages are created at runtime and an alert message is "demoted" as a warning message.

JOLIE **code example**

Listing 58: Deferred Distribution code example

```
include "console.iol"

define pop_alert{
    if(#msg.alert>1){
        for(i=0,i<#msg.alert,i++){
        msg.alert[i]=msg.alert[i+1]
    }};
    undef(msg.alert[#msg.alert-1])
}

define pop_warning{
```

```
12      if(#msg.warning>1){
13          for(i=0,i<#msg.warning,i++){
14          msg.warning[i]=msg.warning[i+1]
15      }};
16      undef(msg.warning[#msg.warning-1])
17  }
18
19  main{
20      for(i=0,i<5,i++){
21          if(i<3){
22              msg.alert[i]="Alert message "+i
23          }
24          else{
25              msg.warning[i-3]="Warning message "+(i-3)
26          }
27      };
28      msg.warning[#msg.warning]="[Demoted] "+msg.alert[#msg.alert-1];
29      undef(msg.alert[#msg.alert-1]);
30      {{
31          synchronized(alert){
32              println@Console("Alert msg: "+msg.alert[0])();
33              pop_alert}
34      }|{for(j=0,j<3,j++){
35          println@Console("Warning msg: "+msg.warning[0])();
36          pop_warning}
37      }|{
38          synchronized(alert){
39              println@Console("Alert msg: "+msg.alert[0])();
40              pop_alert}
41      }}
42  }
```

3.4.4.4 Authorization

The ability to specify the range of privileges that a resource possesses in regard to the execution of a process. In the main, these privileges define the range of actions that a resource can initiate when undertaking work items associated with tasks in a process.

Motivation

Through the specification of authorizations on task definitions, it is possible to define a security framework over a process that is independent of the way in which work items are actually routed at runtime. This can be used to restrict the range of resources that can access details of a work item or request, execute or redistribute it.

This ensures that unexpected events that may arise during execution (e.g. work item delegation by a resource or reallocation to another resource outside of the usual process definition) do not lead to unexpected resources being able to undertake work items.

JOLIE **Implementation**

The Authorization pattern takes the form of a set of relationships between resources and the privileges that they possess in regard to a given process. These privileges define the range of actions that the resource can initiate and can include operations such as:

- *choose* - the ability to select the next work item that they will execute;
- *concurrent* - the ability to execute more than one work item simultaneously;
- *reorder* - the ability to reorder work items in their work list;
- *view offers* - the ability to view all offered work items in the process environment;
- *view allocations* - the ability to view all allocated work items in the process environment;
- *view executions* - the ability to view all executing work items in the process environment;

- *chained execution* - the ability to enter the chained execution mode.

Additionally, it is also possible to specify further user privileges on a per task basis including:

- *suspend* - the ability to suspend and resume instances of this task during execution;
- *stateless reallocate* - the ability to reallocate instances of this task which have been commenced to another user;
- *stateful reallocate* - the ability to reallocate instances of this task which have been commenced to another user and retain any associated state data;
- *deallocate* - the ability to deallocate instances of this task which have not been commenced and allow them to be re-allocated;
- *delegate* - the ability to delegate instances of this task which have not been commenced to another user;
- *skip* - the ability to skip instances of this task;
- *piled execution* - the ability to enter the piled execution mode for work items corresponding to this task.

An offering achieves full support if it satisfies the description for the pattern.

A JOLIE implementation of the *Authorization* pattern can be obtained by providing a specific process, which handles each *Authorization* request. This process is invoked by a resource that sends a specific request to it, on a work item. Among other kind of data sent within the request by the resource, there are 2, in particular, which are used to authenticate the resource and to send, to the work item handler, the permissions linked to the requesting resource. Once authorized the resource can operate allowed instructions on data.

Since a JOLIE code example of this pattern would contain several lines of code, for the sake of brevity no code example is provided for this pattern, instead, after having described how an authorization process structure can be implemented in JOLIE, the available operations on work items, defined previously by the *Authorization* pattern, are taken into account and declined into their corresponding implementation in the language:

- *choose* - the ability to select the next work item that they will execute. This feature is easily implemented by sending a reference to the whole pool of work items (see the *view offers* operation) to the requesting resource, allowing it to subsequently choose the next item among those;
- *concurrent* - the ability to execute more than one work item simultaneously. This operation has an easy implementation too. In this case the ability to execute concurrent operations on multiple work item is obtained by letting the resource access several work item, without expecting any output corresponding to the end of the activity of that resource on a single work item;
- *reorder* - the ability to reorder work items in their work list. As a slight modification of the *choose* pattern described above, the *reorder* pattern lets access the resource to the whole pool of its work items, after having obtained the list, the resource can reorder their sequence according to its needs, and send back the new disposition to the work item handler;
- *view offers* - the ability to view all offered work items in the process environment. As stated before, this is a fundamental operation which is achieved by sending to the requesting resource the full (list) structure of work items. Scoped views can be easily obtained by sub-structuring the work items list according to scope necessities.
- *view allocations* - the ability to view all allocated work items in the process environment. This operation is based on a work item state log structure, which keeps track of each work item operation, among which is the allocation one.
- *view executions* - the ability to view all executing work items in the process environment. The same approach described for the *view allocations* pattern can be used for the *view executions* to show all the executing work items in the environment.

Further operations like *suspend* [3.4.7.6], *stateless reallocate* [3.4.7.5] , *stateful reallocate* [3.4.7.4] , *deallocate* [3.4.7.3] , *delegate* [3.4.7.1] , *skip* [3.4.7.7] and *piled execution* [3.4.8.3] are described and analyzed separately as it follows, since their behavior can be allowed as a consequence of an *Authorization* construct, but they do not necessarily be preceded by such a construct to operate properly.

3.4.4.5 Separation of Duties

The ability to specify that two tasks must be executed by different resources in a given case.

Motivation

Separation of Duties allows for the enforcement of audit controls within the execution of a given case. The *Separation of Duties* constraint exists between two tasks in a process model. It ensures that within a given case, work items corresponding to the latter task cannot be executed by resources that completed work items corresponding to the former task. Another use of this pattern arises with multiple task instances. In this situation, the degree of parallelism that can be achieved when a multiple instance task is executed can be maximized by specifying that as far as possible no two task instances can be executed by the same resource.

JOLIE **Implementation**

The *Separation of Duties* pattern relates a task to a number of other tasks that precede it in the process. Within a given case, work items corresponding to task cannot be distributed to any resource that previously completed work items corresponding to tasks with which it has a *Separation of Duties* constraint. As it is possible that preceding tasks may have executed more than once within a given case, e.g. they may be contained within a loop or have multiple instances, there may be a number of resources that are excluded from undertaking instances of that task.

An offering achieves full support if it satisfies the description for the pattern. It achieves a partial support rating where the same effect can be achieved indirectly, e.g. using access rights on tasks or security constraints.

A JOLIE implementation of the *Separation of Duties* pattern can be obtained by using dedicated values, linked to the item, which are used to state the identity (differentially/indirectly or directly) of the resource enabled to work on that specific item.

As stated in support definition, this implementation achieves a partial support for this pattern.

JOLIE **code example**

Listing 59: Separation of Duties code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define add_new_wi{
    random@Math()(id);
    synchronized(lock){
        wi[#wi].idd=int(id*100000)
    };
    sleep@Time(2001)()
}

main{
    {while(true){
        add_new_wi
    }}|{
        while(true){
            synchronized(lock){
                if(#wi>1 && is_defined(wi[#wi-1])){
                    if(!is_defined(wi[#wi-1].preproc)){
                        rid1.preproc.wi<<wi[#wi-1];
                        undef(wi[#wi-1])
                    }
                    else if(!is_defined(wi[#wi-1].proc) &&
                        wi[#wi-1].preproc!="rid1"){
                        rid1.proc.wi<<wi[#wi-1];
                        undef(wi[#wi-1])
                    }
            }};
            if (is_defined(rid1.preproc.wi)){
                sleep@Time(2000)();
                println@Console("RID1 Preprocessed "+
                    "work item id: "+rid1.preproc.wi.idd)();
                rid1.preproc.wi.preproc="rid1";
                synchronized(lock){
                    wi[#wi]<<rid1.preproc.wi
                };
                undef(rid1.preproc.wi)
            }
            else if (is_defined(rid1.proc.wi)){
                sleep@Time(1000)();
                println@Console("RID1 Processed "+
                    "work item id: "+rid1.proc.wi.idd)();
                rid1.proc.wi.proc="rid1";
                synchronized(lock){
                    wi[#wi]<<rid1.proc.wi
                };
                undef(rid1.proc.wi)
```

```
            }
    }}|{
        while(true){
            synchronized(lock){
                if(#wi>1 && is_defined(wi[#wi-1])){
                    if(!is_defined(wi[#wi-1].preproc)){
                        rid2.preproc.wi<<wi[#wi-1];
                        undef(wi[#wi-1])
                    }
                    else if(!is_defined(wi[#wi-1].proc) &&
                        wi[#wi-1].preproc!="rid2"){
                        rid2.proc.wi<<wi[#wi-1];
                        undef(wi[#wi-1])
                    }
            }};
            if (is_defined(rid2.preproc.wi)){
                sleep@Time(2000)();
                println@Console("RID2 Preprocessed "+
                    "work item id: "+rid2.preproc.wi.idd)();
                rid2.preproc.wi.preproc="rid2";
                synchronized(lock){
                    wi[#wi]<<rid2.preproc.wi
                };
                undef(rid2.preproc.wi)
            }
            else if (is_defined(rid2.proc.wi)){
                sleep@Time(1000)();
                println@Console("RID2 Processed "+
                    "work item id: "+rid2.proc.wi.idd)();
                rid2.proc.wi.proc="rid2";
                synchronized(lock){
                    wi[#wi]<<rid2.proc.wi
                };
                undef(rid2.proc.wi)
            }
        }
    }

}
```

3.4.4.6 Case Handling

The ability to allocate the work items within a given case to the same resource at the time that the case is commenced.

Motivation

Case Handling is a specific approach to work distribution that is based on the premise that all work items in a given case are so closely related that they should all be undertaken by the same resource. The identification of the specific resource occurs when a case (or the first work item in a case) requires allocation.

Case Handling may occur on either a "hard" or "soft" basis i.e. work items within a given case can be allocated exclusively to the same resource which must complete them all or alternatively it can serve as a guide to how work items within a given case should be routed with an initial resource being identified as having responsibility for all work items and subsequently delegating them to other resources or allowing them to nominate work items they would like to complete.

JOLIE **Implementation**

The *Case Handling* pattern takes the form of a relationship between a process and one or more resources or roles. When an instance of the process is initiated, a resource is selected from the set of resources and roles and the process instance is allocated to this resource. It is expected that this resource will execute work items corresponding to tasks in this process instance.

There are no specific context conditions associated with this pattern. An offering achieves full support if it satisfies the description for the pattern.

A JOLIE implementation of the *Case Handling* pattern can be obtained as a slight modification of the one provided for the *Separation of Duties* pattern [3.4.4.5]. In this case the dedicated value, linked to the item, identifies uniquely the set (whose cardinality equals 1, in the example) of resources enabled to work on that specific item.

Given the particular similarity of this implementation and the *Separation of Duties* only a brief snippet of code, reporting the relevant modifications from that example is provided as it follows.

It's worth noting that, even if the *Separation of Duties* example is based on the separation of work items among different resources, this example can be easily mod-

ified to realize an instance(case)-based separation behavior, by defining a server-client structure and a concurrent execution of the server, which starts a new case for each client invocation (case creation).

JOLIE **code example**

Listing 60: Case Handling code example

```
                    else if(!is_defined(wi[#wi-1].proc) &&
                        wi[#wi-1].preproc=="rid1"){
                        rid1.proc.wi<<wi[#wi-1];
                        undef(wi[#wi-1])
                    }
            [...]

                    else if(!is_defined(wi[#wi-1].proc) &&
                        wi[#wi-1].preproc=="rid2"){
                        rid2.proc.wi<<wi[#wi-1];
                        undef(wi[#wi-1])
                    }
            [...]
```

3.4.4.7 Retain Familiar

Where several resources are available to undertake a work item, the ability to allocate a work item within a given case to the same resource that undertook a preceding work item.

Motivation

Distributing a work item to the same resource that undertook a previous work item is a common means of expediting a case. As the resource is already aware of the details of the case, it saves familiarization time at the commencement of the work item. Where the two work items are sequential, it also offers the opportunity for minimizing switching time as the resource can commence the latter work item immediately on completion of the former.

This pattern is a more flexible version of the *Case Handling* pattern [3.4.4.6] discussed earlier.

It only comes into effect when there are multiple resources available to undertake a given work item and where this occurs, it favors the allocation of the work item to the resource that undertook a previous work item in the case. Unlike the *Case Handling* pattern (which operates at case level), this pattern applies at the work item level and comes into play when a work item is being distributed to a resource.

The *Chained Execution* pattern [3.4.8.4] is related to this pattern and is designed to expedite the completion of a case by automatically starting subsequent work items once the preceding work item is complete.

JOLIE **Implementation**

The *Retain Familiar* pattern takes the form of a one-one relationship between a task and a preceding task in the same process. Where it holds for a task, when an instance of the task is created in a given case, it is distributed to one of the nominated resources the completed one of the preceding tasks in the same case. If the preceding task has been executed more than once, it is distributed to one of the resources that completed it previously.

There are no specific context conditions associated with this pattern.

A JOLIE implementation of the *Retain Familiar* pattern can be obtained as a slight modification of the ones provided for the *Separation of Duties* [3.4.4.5] pattern and *Case Handling* pattern. In this case multiple tasks on each item can be operated only by the resource which operated the first task on that item.

3.4.4.8 Capability-Based Distribution

The ability to distribute work items to resources based on specific capabilities that they possess. Capabilities (and their associated values) are recorded for individual resources as part of the organizational model.

Motivation

Capability-based Distribution provides a mechanism for offering or allocating work items to resources through the matching of specific requirements of work items with the capabilities of the potential range of resources that are available to undertake them. This allows for a much more fine-grained approach to selecting the resources suitable for completing a given task.

JOLIE **Implementation**

Within a given organizational model, each resource is assumed to be able to have capabilities recorded for them that specify their individual characteristics (e.g. qualifications, previous jobs) and their ability to undertake certain tasks (e.g. licenses held, trade certifications).

Similarly it is assumed that capability functions can be specified that take a set of resources and their associated capabilities and return the subset of those resources that conform to a required range of capability values. Each task in a process model can have a capability function associated with it.

Capability-based Distribution can be either push or pull-based, i.e. the actual distribution process can be initiated by the system or the resource. In the former situation, the system determines the most appropriate resource(s) to which a work item should be routed. In the latter, a resource initiates a search for an unallocated work item(s) which it is capable of undertaking.

Capability-based Distribution is based on the specification of capabilities for individual resources. Capabilities generally take the form of attribute-value pairs (e.g. "signing authority", "$10M"). A dictionary of capabilities can be defined in which individual capabilities have a distinct name and the type and potential range of values that each capability may take can also be specified. Similarly, tasks can also have capabilities recorded for them.

The actual distribution process is generally based on the specification of functions which are evaluated at runtime and determine how individual work items can be matched with suitable resources. These may be arbitrarily complex in nature

depending on the range of capabilities that require matching between resources and work items and the approach that is taken to ranking the matches that are achieved in order to select the most appropriate resource to undertake a given work item.

An offering achieves full support if it satisfies the description of the pattern.

The JOLIE implementation of the *Capability-based Distribution* pattern is based on a pull approach. Each resource checks periodically the presence of items whose requirements are satisfiable by its capabilities and once found, the item is withdrawn by the resource, which finally processes it.

In this particular example, a branch is constantly creating new processes whose requirement are random and divided on 2 different requirements: an architectural one (x86, x64) and prioritizing one (from 0 to 10). Along with the creation branch there are 3 other branches: two (RID1 RID2) which have the ability to run processes of any priority, but only of one specific architecture, contrariwise the branch RID3 can perform any process, regardless of the architecture, but of low priority.

JOLIE **code example**

Listing 61: Capability-based Distribution code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define add_new_wi{
    random@Math()(id);
    random@Math()(proc_priority);
    random@Math()(proc_arch);
    synchronized(lock){
        newwi.id=int(id*10000);
        proc_arch=int(proc_arch*2);
        if(proc_arch>0){
            newwi.proc_arch="x86"
        }
        else{
            newwi.proc_arch="x64"
        };
        newwi.proc_priority=int(proc_priority*11);
        wi[#wi]<<newwi
    };
    sleep@Time(500)()
}

```

```
define print_wi{
    string_wi="Wi queue: {"+
        wi[0].id+"|"+
        wi[0].proc_arch+"|"+
        wi[0].proc_priority+"}";
    for(j=1,j<#wi,j++){
        string_wi=string_wi+",{"+
        wi[j].id+"|"+
        wi[j].proc_arch+"|"+
        wi[j].proc_priority+"}"
    };
    println@Console(string_wi)();
    undef(string_wi)
}

define pop_wi{
    print_wi;
    if(#wi>0){
        for(pop.i=pop_wi_n,pop.i<(#wi-1),pop.i++){
            wi[pop.i]<<wi[(pop.i+1)]
    };
    undef(wi[#wi-1])
    }
}

main{
    {while(true){
        add_new_wi
    }}|{
        while(true){
            synchronized(lock){
                for(rid1.i=0,rid1.i<#wi,rid1.i++){
                    if(wi[rid1.i].proc_arch=="x86"){
                        println@Console("RID1 Found "+
                            wi[rid1.i].id+"|"+
                            wi[rid1.i].proc_arch+"|"+
                            wi[rid1.i].proc_priority)();
                        rid1.wi<<wi[rid1.i];
                        pop_wi_n=rid1.i;
                        pop_wi;
                        rid1.i=#wi
                    }
            }};
            if(is_defined(rid1.wi)){
                println@Console("RID1 Processing"+
                          " wi: "+rid1.wi.id)();
                sleep@Time(2000)();
                undef(rid1.wi)
            }
    }}|{
        while(true){
            synchronized(lock){
```

```
76                  for(rid2.i=0,rid2.i<#wi,rid2.i++){
77                      if(wi[rid2.i].proc_arch=="x64"){
78                          println@Console("RID2 Found "+
79                              wi[rid2.i].id+"|"+
80                              wi[rid2.i].proc_arch+"|"+
81                              wi[rid2.i].proc_priority)();
82                          rid2.wi<<wi[rid2.i];
83                          pop_wi_n=rid2.i;
84                          pop_wi;
85                          rid2.i=#wi
86                      }
87              }};
88              if(is_defined(rid2.wi)){
89                  println@Console("RID2 Processing"+
90                              " wi: "+rid2.wi.id)();
91                  sleep@Time(2000)();
92                  undef(rid2.wi)
93              }
94      }}|{
95          while(true){
96              synchronized(lock){
97                  for(rid3.i=0,rid3.i<#wi,rid3.i++){
98                      if(wi[rid3.i].proc_priority<6){
99                          println@Console("RID 3 Found "+
100                             wi[rid3.i].id+"|"+
101                             wi[rid3.i].proc_arch+"|"+
102                             wi[rid3.i].proc_priority)();
103                         rid3.wi<<wi[rid3.i];
104                         pop_wi_n=rid3.i;
105                         pop_wi;
106                         rid3.i=#wi
107                     }
108             }};
109             if(is_defined(rid3.wi)){
110                 println@Console("RID3 Processing"+
111                             " wi: "+rid3.wi.id)();
112                 sleep@Time(2000)();
113                 undef(rid3.wi)
114             }
115     }}
116 }
```

3.4.4.9 History-Based Distribution

The ability to distribute work items to resources on the basis of their previous execution history.

Motivation

History-based Distribution involves the use of information on the previous execution history of resources when determining which of them a work item should be distributed to. This is an analogue to common human experience when determining who to distribute a specific work item to which considers factors such as who has the most experience with this type of work item or who has had the least numbers of failures when tackling similar tasks.

JOLIE **Implementation**

History-based Distribution assumes the existence of historical distribution functions which take a set of resources and the previous execution history for the process and return the subset of those resources that satisfy the nominated historical criteria. These may include factors such as the resource that least recently executed a task, has executed it successfully the most times, has the shortest turnaround time for the task or any other combination of requirements that can be determined from the execution history. Each task in a process model can have a historical distribution function associated with it.

There are no specific context conditions associated with this pattern. An offering achieves full support if it satisfies the description for the pattern. It achieves a partial support rating if the same effect can be achieved via programmatic extensions.

The JOLIE implementation of the *History-based Distribution* pattern is provided by means of two parallel processes: the "create" one whose task is to create new work items, that represent processes which shall be run on a specific execution architecture environment, as a matter of facts, the resources available in the system. Along with this branch, there's an "execute" process that is invoked by means of a self-RequestResponse operation. Each invocation is made by the "create" process, which checks, chooses and updates each resource's execution data, according to the chosen environment.

Each resource has a different process architecture execution multiplier, which, randomly, defines the performance of any resource execution, according to the work item architecture. Each "work item architecture - resource execution time" couple

is used by the "create" process to update a global resource performances table, whose calculation is set as a weighted average among resources executions. This data is subsequently used by the "create" process to assign to the specific (optimal) resource a work item, by means of its execution architecture.

JOLIE **code example**

Listing 62: History-based Distribution code example

```
include "console.iol"
include "time.iol"
include "math.iol"

inputPort HBD{
    Location: "socket://localhost:8000"
    Protocol: sodep
    RequestResponse: execute_rid
}

outputPort HBD{
    Location: "socket://localhost:8000"
    Protocol: sodep
    RequestResponse: execute_rid
}

define create_wi{
    random@Math()(arch);
        arch=int(arch*3);
        if(arch<1){
            wi.arch="x86"
        }
        else if(arch<2){
            wi.arch="x64"
        }else{
            wi.arch="arm"
        }
}

define print_rid_stats{
    println@Console("---=== RIDs STATS (avgs) ===---")();
    for(i=0,i<3,i++){
        println@Console("RID"+(i)+":")();
        println@Console("\tx86: "+rid[i].x86_runtime+" | runned "+
            rid[i].x86_run)();
        println@Console("\tx64: "+rid[i].x64_runtime+" | runned "+
            rid[i].x64_run)();
        println@Console("\tarm: "+rid[i].arm_runtime+" | runned "+
            rid[i].arm_run)()
    };
    println@Console("---=========================---")()
}

init{
```

```
    for(i=0,i<3,i++){
        rid[i].x86_runtime=0;rid[i].x86_run=0;
        rid[i].x64_runtime=0;rid[i].x64_run=0;
        rid[i].arm_runtime=0;rid[i].arm_run=0
    };
    rid[0].x86.exec_mult=1;rid[0].exec_mult.x64=2;rid[0].exec_mult.arm=3;
    rid[1].x86.exec_mult=3;rid[1].exec_mult.x64=1;rid[1].exec_mult.arm=2;
    rid[2].x86.exec_mult=2;rid[2].exec_mult.x64=3;rid[2].exec_mult.arm=1
}

main{
    {while(true){
        //create a new wi
        create_wi;
        println@Console("Created new wi, wi arch: "+wi.arch)();
        //print stats
        print_rid_stats;
        //choose RID exec
        rid_exec.time=100000000;
        for(i=0,i<3,i++){
            if(rid_exec.time>rid[i].(wi.arch+"_runtime")){
                rid_exec.rid=i;
                rid_exec.time=rid[rid_exec.rid].(wi.arch+"_runtime")
            }
        };
        rid_data.wi<<wi;
        rid_data.rid=rid_exec.rid;
        execute_rid@HBD(rid_data)(run_time);
        rid[rid_exec.rid].(wi.arch+"_runtime")=
            (rid[rid_exec.rid].(wi.arch+"_runtime")*
                rid[rid_exec.rid].(wi.arch+"_run")+run_time)/
                    (rid[rid_exec.rid].(wi.arch+"_run")+1);
        rid[rid_exec.rid].(wi.arch+"_run")++
    }}|{
        while(true){
            execute_rid(rid_data)(run_time){
            random@Math()(exec_time);
            run_time=int(exec_time*(rid[rid_data.rid].(
                rid_data.wi.arch).exec_mult*500));
            sleep@Time(run_time)()
            }
        }
    }
}
```

3.4.4.10 Organizational Distribution

The ability to distribute work items to resources based their position within the organization and their relationship with other resources.

Motivation

Most offerings provide some degree of support for modeling the organizational context in which a given process operates. This is an important aspect of business process modeling and implementation as many work distribution decisions are made in the context of the organizational structure and the relative position of individual resources both in the overall hierarchy and also in terms of their relationships with other resources. The ability to capture and emulate these types of work distribution strategies are an important requirement to provide a flexible and realistic basis for managing work in an organizational setting.

JOLIE **Implementation**

Organizational Distribution assumes the existence of organizational distribution functions which take a set of resources and the organizational model associated with a process and return the subset of those resource that satisfy the nominated organizational criteria. These may include factors such as members of a specified department, resources holding a certain position, resources that report to a nominated individual or any other combination of requirements that can be determined from the organizational model. Each task in a process model can have an organizational distribution function associated with it.

An offering achieves full support if it satisfies the description for the pattern. It achieves a partial support rating if the same effect can be achieved via programmatic extensions.

The JOLIE implementation of the *Organizational Distribution* pattern is provided by means of a process handler executed in concurrent behavior with the non-deterministic choice construct. Such process is used to redirect the incoming work items towards the corresponding organizational resource.

In this example is represented a process whose task is to redirect (allocate) media processing data towards the corresponding resource that can handle that type of item. Furthermore, a reporting (log) process, withing the organizational structure, is run parallelly for logging purposes.

JOLIE **code example**

Listing 63: Organizational Distribution code example

```
include "console.iol"
include "time.iol"
include "math.iol"

inputPort OD{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: proc_handler, proc_image, proc_video,
        proc_audio, proc_log, start
}

outputPort OD{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: proc_handler, proc_image, proc_video,
        proc_audio, proc_log, start
}

init{
    with (global.log){
        .audio=0;
        .video=0;
        .image=0
    };
    start@OD()
}

execution{concurrent}

define create_wi{
    random@Math()(type);
    with(wi){
        .image=false;.video=false;.audio=false
    };
        type=int(type*3);
        if(type<1){
            wi.image=true
        }
        else if(type<2){
            wi.video=true
        }else{
            wi.audio=true
        };
        wi.task=true;
    proc_handler@OD(wi);
    sleep@Time(500)()

}

define print_log{
    println@Console("---=== PROCESSED DATA LOG ===---")();
```

```
52      println@Console("\timages: "+global.log.image)();
53      println@Console("\tvideos: "+global.log.video)();
54      println@Console("\taudios: "+global.log.audio)();
55      println@Console("---============================---")()
56  }
57
58  main{
59      [start()]{
60          while(true){
61              create_wi
62          }
63      }
64      //PROC HANDLER
65      [proc_handler(proc_data)]{
66      if(proc_data.task){
67          if(proc_data.image){
68              proc_image@OD(proc_data)
69          }
70          else if(proc_data.video){
71              proc_video@OD(proc_data)
72          }
73          else if(proc_data.audio){
74              proc_audio@OD(proc_data)
75          }
76      }
77      else if(proc_data.log){
78          proc_log(proc_data)
79      }}
80      [proc_image(wi)]{
81          sleep@Time(1000)();
82          wi.log=true;
83          wi.task=false;
84          proc_log@OD(wi)
85      }
86      [proc_audio(wi)]{
87          sleep@Time(2000)();
88          wi.log=true;
89          wi.task=false;
90          proc_log@OD(wi)
91      }
92      [proc_video(wi)]{
93          sleep@Time(3000)();
94          wi.log=true;
95          wi.task=false;
96          proc_log@OD(wi)
97      }
98      [proc_log(wi)]{
99          if(wi.image){
100             global.log.image++
101         }else if(wi.audio){
102             global.log.audio++
```

```
103             }else if(wi.video){
104                 global.log.video++
105             };
106             print_log
107         }
108     }
```

3.4.4.11 Automatic Execution

The ability for an instance of a task to execute without needing to utilize the services of a resource.

Motivation

Not all tasks within a process need to be executed under the auspices of a human resource, some are able to execute independently once the specified enabling criteria are met.

JOLIE **Implementation**

Where a task is nominated as automatic, it is initiated immediately when enabled. Similarly, upon its completion, subsequent tasks are triggered immediately.

An offering achieves full support if it satisfies the description for the pattern. It achieves a partial support rating if the same effect can be achieved via programmatic extensions.

JOLIE implements natively this behavior by letting the task process to run (both sequentially or parallelly) without requiring its distribution to a specific resource.

3.4.5 Push Patterns

Push Patterns characterize situations where newly created work items are proactively offered or allocated to resources by the system. These may occur indirectly by advertising work items to selected resources via a shared work list or directly with work items being allocated to specific resources.

In both situations however, it is the system that takes the initiative and causes the distribution process to occur.

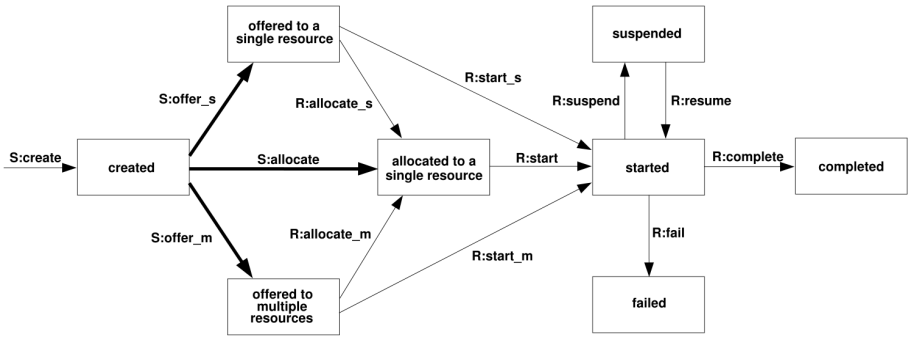


Figure 3.46: Push Patterns

As Figure 3.46 reports, nine push Patterns have been identified. These are divided into three distinct groups.

- The first three Patterns identify the actual manner of work distribution - whether the workflow system offers the work item to a single resource, to multiple resources or whether it allocates it directly to a single resource. These patterns correspond directly to the bold arcs in the figure.
- The second group of patterns relate to the means by which a resource is selected to undertake a work item where there are multiple possible resources identified. Three possible strategies are described - random allocation, round robin allocation and shortest queue.
- The final three patterns identify the timing of the distribution process and in particular the relationship between the availability of a work item for offering/allocation to resources and the time at which it commences execution. Three variants are possible - work items are offered/allocated before they

have commenced (early distribution), after they have commence (late distribution) or the two events are simultaneous (distribution on enablement). These Patterns do not have a direct analogue in the figure but relate to the time at which the transitions may occur with respect to the work item's readiness to be executed (i.e. already started, immediate start or subsequent start).

3.4.5.1 Distribution by Offer - Single Resource

The ability to distribute a work item to a selected individual resource on a non-binding basis.

Motivation

This pattern provides a means of distributing a work item to a single resource on a non-binding basis. The resource is informed of the work item being offered but it is not committed to execute it and can either ignore the work item or redistribute it to other resources should it choose not to undertake it.

JOLIE **Implementation**

Offering a work item to a single resource is the process analogy to the act of "asking for consideration" in real life. If the resource decides not to undertake it, the onus is still with the system to find another suitable resource to complete it. Once a task has been enabled, a means of actually informing the selected resource of the pending work item is required. The mechanism chosen, should notify the resource that a work item exists that it may wish to undertake, however it should not commit the resource to its execution and it should not advise any other resources of the potential work item. Typically this is achieved by adding the work item to the work list of the selected user with an offered status although other notification mechanisms are possible.

An offering achieves full support it is satisfies the description for the pattern. It achieves a partial support rating if work items cannot be distributed on a non-binding basis but there are facilities for a resource to reject a work item allocated to it.

The JOLIE implementation of the *Distribution by Offer - Single Resource* pattern is provided by means of a process which creates and offers work items to a specific

resource (adding the work item to the work items list of that resource) among the three available (RID1, RID2 and RID3).

Each resource can either decide (randomly) to process that work item or re-offer it to it's "co-worker". In this example the passing policy is defined at design time where each resource has a designed co-worker to which it can offer its work item. It's worth noting that the passing relation between them is intentionally recursive, i.e. RID1 can offer to RID2, that can offer to RID3 which that can offer to RID1 and so on.

JOLIE **code example**

Listing 64: Distribution by Offer - Single Resource code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define pop_rid1{
    if(#wi_stack.rid1>1){
        for(rid1.i=0,rid1.i<#wi_stack.rid1,rid1.i++){
        wi_stack.rid1[rid1.i]=wi_stack.rid1[rid1.i+1]
    }};
    undef(wi_stack.rid1[#wi_stack.rid1-1])
}

define pop_rid2{
    if(#wi_stack.rid2>1){
        for(rid2.i=0,rid2.i<#wi_stack.rid2,rid2.i++){
        wi_stack.rid2[rid2.i]=wi_stack.rid2[rid2.i+1]
    }};
    undef(wi_stack.rid2[#wi_stack.rid2-1])
}

define pop_rid3{
    if(#wi_stack.rid3>1){
        for(rid3.i=0,rid3.i<#wi_stack.rid3,rid3.i++){
        wi_stack.rid3[rid3.i]=wi_stack.rid3[rid3.i+1]
    }};
    undef(wi_stack.rid3[#wi_stack.rid3-1])
}

define create_and_offer_wi{
    random@Math()(rid);
    rid=1+int(rid*3);
    getCurrentTimeMillis@Time()(id);
    synchronized(wi_lock){
        wi_stack.("rid"+rid)[#wi_stack.("rid"+rid)]=id
    };
    println@Console("Created wi "+id+" and offered to RID"+rid)();
    sleep@Time(500)()
```

```
}

main{
    {while(true){
        create_and_offer_wi
    }}|{
    while(true){
        if(#wi_stack.rid1>0){
            random@Math()(rid1.deleg);
            rid1.deleg=int(rid1.deleg*2);
            if(rid1.deleg){
                synchronized(wi_lock){
                    wi_stack.rid2[#wi_stack.rid2]=wi_stack.rid1[0];
                    println@Console("RID1 offered the wi "+wi_stack.rid1[0]+
                        " to RID2")();
                    sleep@Time(500)()
                }
            }
            else{
                println@Console("RID1 processes wi "+wi_stack.rid1[0])();
                sleep@Time(1000)()
            };
            synchronized(wi_lock){
                pop_rid1
            }}
        }
    }|{
    while(true){
        if(#wi_stack.rid2>0){
            random@Math()(rid2.deleg);
            rid2.deleg=int(rid2.deleg*2);
            if(rid2.deleg){
                synchronized(wi_lock){
                    wi_stack.rid3[#wi_stack.rid3]=wi_stack.rid2[0];
                    println@Console("RID2 offered the wi "+wi_stack.rid2[0]+
                    " to RID3")();
                    sleep@Time(500)()
                }
            }
            else{
                println@Console("RID2 processes wi "+wi_stack.rid2[0])();
                sleep@Time(1000)()
            };
            synchronized(wi_lock){
                pop_rid2
            }}
        }
    }|{
    while(true){
        if(#wi_stack.rid3>0){
            random@Math()(rid3.deleg);
```

```
89              rid3.deleg=int(rid3.deleg*2);
90              if(rid3.deleg){
91                  synchronized(wi_lock){
92                      wi_stack.rid1[#wi_stack.rid1]=wi_stack.rid3[0];
93                      println@Console("RID3 offered the wi "+wi_stack.rid3[0]+
94                      " to RID1")();
95                      sleep@Time(500)()
96                  }
97              }
98              else{
99                  println@Console("RID3 processes wi "+wi_stack.rid3[0])();
100                 sleep@Time(1000)()
101             };
102             synchronized(wi_lock){
103                 pop_rid3
104             }}
105         }
106     }
107 }
```

3.4.5.2 Distribution by Offer - Multiple Resources

The ability to distribute a work item to a group of selected resources on a non-binding basis.

Motivation

This pattern provides a means of distributing a work item to multiple resources on a non-binding basis. The resources are informed of the work item being offered but are not committed to executing it and can either ignore the work item or redistribute it to other resources should they choose not to undertake it.

JOLIE **Implementation**

Offering a work item to multiple resources is the process analogy to the act of "calling for a volunteer" in real life. It provides a means of advising a suitably qualified group of resources that a work item exists with the expectation that one of them will actually commit to undertaking the activity although the onus is still with the system to find a suitable resource should none of them agree to undertake it. Once a task has been enabled that is distributed on this basis, a means of actually informing the selected resources of the pending work item is required. The mechanism chosen, should notify the resources that a work item exists that they may wish to undertake, however it should not commit any of the resources to its execution. Typically this is achieved by adding the work item to the work lists of the selected resources with an offered status although other notification mechanisms are possible.

An offering achieves full support if it satisfies the description of the pattern.

The JOLIE implementation of the *Distribution by Offer - Multiple Resources* pattern is provided by means of a slight modification of the one given for the *Distribution by Offer - Multiple Resources* pattern [3.4.5.1].

In this case the process which creates and offers work items to the resources simply adds the work item to a shared work item list that corresponds to a group of resources. In this example the three available resources (RID1, RID2 and RID3) share two lists of work items (one read by RID1 and RID2, one read by RID2 and RID3).

In this particular example RID2 has the opportunity to choose among two lists of work items and decide to process the first available work item of any of them.

JOLIE **code example**

Listing 65: Distribution by Offer - Multiple Resource code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define pop_lst1{
    if(#wi.lst1>1){
        for(l1i=0,l1i<#wi.lst1,l1i++){
        wi.lst1[l1i]=wi.lst1[l1i+1]
    }};
    undef(wi.lst1[#wi.lst1-1])
}

define pop_lst2{
    if(#wi.lst2>1){
        for(l2i=0,l2i<#wi.lst2,l2i++){
        wi.lst2[l2i]=wi.lst2[l2i+1]
    }};
    undef(wi.lst2[#wi.lst2-1])
}

define create_and_offer_wi{
    random@Math()(lstid);
    lstid=1+int(lstid*2);
    getCurrentTimeMillis@Time()(id);
    synchronized(lst_lock){
        wi.("lst"+lstid)[#wi.("lst"+lstid)]=id
    };
    println@Console("Created wi "+id+" and offered to list"+lstid)();
    sleep@Time(500)()
}

main{
    {while(true){
        create_and_offer_wi
    }}|{
    while(true){
        synchronized(lst_lock){
        if(#wi.lst1>0){
            println@Console("RID1 has taken the wi "+wi.lst1[0])();
            pop_lst1
        }};
        sleep@Time(500)()
    }}|{
    while(true){
        synchronized(lst_lock){
        if (#wi.lst1>0){
            println@Console("RID2 has taken the wi "+wi.lst1[0]+
                " from list1")();
            pop_lst1
```

```
50          }
51          else if(#wi.lst2>0){
52              println@Console("RID2 has taken the wi "+wi.lst2[0]+
53                  " from list2")();
54              pop_lst2
55          }};
56          sleep@Time(500)()
57      }}|{
58      while(true){
59          synchronized(lst_lock){
60          if(#wi.lst2>0){
61              println@Console("RID3 has taken the wi "+wi.lst2[0])();
62              pop_lst2
63          }};
64          sleep@Time(500)()
65      }}
66  }
```

3.4.5.3 Distribution by Allocation - Single Resource

The ability to distribute a work item to a specific resource for execution on a binding basis.

Motivation

This pattern provides a means of distributing a work item to a single resource on a binding basis. The resource is informed of the work item being distributed to them and is committed to executing it.

JOLIE **Implementation**

Allocating a work item to a single resource is the process analogy to the act of "appointing an owner" in real life. It involves the system directly assigning a work item to a resource without first offering it to other resources or querying whether the resource will undertake it. In doing so, it passes the onus of ensuring the work item is completed to the selected resource.

This approach to work distribution is also known as "heads down" processing as it offers the resource little or no input in the work that they are allocated and the main focus is on maximizing work throughput by keeping the resource busy. In many implementations, resources are simply allocated a new work item once the previous one is completed and they are not offered any insight into what work items might lay ahead for them.

Once a task has been enabled that is distributed on this basis, a means of actually informing the selected resource of the pending work item is required. The mechanism chosen, should notify the resource that a work item exists that they must undertake. Typically this is achieved by adding the work item to the work list of the selected resource with an allocated status although other notification mechanism are possible.

An offering achieves full support if it satisfies the description of the pattern.

The JOLIE implementation of the *Distribution by Allocation - Single Resource* pattern is the same given for the *Direct Distribution* pattern [3.4.4.1] where a work item is specifically distributed (allocated) to a resource.

3.4.5.4 Random Allocation

The ability to allocate work items to a selected resource chosen from a group of eligible resources on a random basis.

Motivation

Random Allocation provides a non-deterministic mechanism for allocating work items to resources.

JOLIE **Implementation**

This pattern provides a means of restricting the distribution of a work item to a single resource. Once the possible range of resources that a work item can be distributed to have been identified at runtime, one of these is selected at random to execute the work item.

An offering achieves full support if it satisfies the description for the pattern.

The JOLIE implementation of the *Random Allocation* pattern is provided by means of a shared list of available work items, when a resource is ready to undertake a new work item, it picks one of them randomly.

JOLIE **code example**

Listing 66: Random Allocation code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define pop_wi{
    if(#wi>0){
        for(pop.i=pop_wi_n,pop.i<(#wi-1),pop.i++){
            wi[pop.i]<<wi[(pop.i+1)]
    };
    undef(wi[#wi-1])
}}

define create_and_offer_wi{
    getCurrentTimeMillis@Time()(id);
    synchronized(wi_lock){
        wi[#wi]=id
    };
    println@Console("Created wi "+id)();
    sleep@Time(200)()
}

main{
```

```
23      {while(true){
24          create_and_offer_wi
25      }}|{
26      while(true){
27          synchronized(wi_lock){
28              if(#wi>0){
29                  random@Math()(wid);
30                  wid=int(wid*#wi);
31                  println@Console("RID1 has taken the wi "+wi[wid])();
32                  pop_wi_n=wid;
33                  pop_wi
34              }};
35          sleep@Time(500)()
36      }}|{
37      while(true){
38          synchronized(wi_lock){
39              if(#wi>0){
40                  random@Math()(wid);
41                  wid=int(wid*#wi);
42                  println@Console("RID2 has taken the wi "+wi[wid])();
43                  pop_wi_n=wid;
44                  pop_wi
45              }};
46          sleep@Time(500)()
47      }}|{
48      while(true){
49          synchronized(wi_lock){
50              if(#wi>0){
51                  random@Math()(wid);
52                  wid=int(wid*#wi);
53                  println@Console("RID3 has taken the wi "+wi[wid])();
54                  pop_wi_n=wid;
55                  pop_wi
56              }};
57          sleep@Time(500)()
58      }}
59  }
```

3.4.5.5 Round Robin Allocation

The ability to allocate a work item to a selected resource chosen from a group of eligible resources on a cyclic basis.

Motivation

Round Robin Allocation provides a means of allocating work items to resources on an equitable basis.

JOLIE **Implementation**

This pattern provides a fair means of restricting the distribution of a work item to a single resource. Once the range of possible resources that a work item can be distributed to has been identified at runtime, one of these is selected on a cyclic basis to execute the work item. The intention is that, over time, each resource receives the same number of work items. One means of choosing the appropriate resource is to select the resource that undertook the task least recently. An alternative to this is for the system to keep track of the number of times each resource has completed each task, thus enabling the one who has undertaken it the least number of times to be identified.

An offering achieves full support if it satisfies the description for the pattern.

A simple JOLIE implementation of the *Round Robin Allocation* pattern is provided by means of a cyclic list of resource on which the resource creator allocates a work item to a specific resource.

JOLIE **code example**

Listing 67: Round Robin code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define create_and_allocate_wi{
    getCurrentTimeMillis@Time()(id);
    rid++;
    if(rid>3){rid=1};
    synchronized(wi_lock){
        wi.("rid"+rid)[#wi.("rid"+rid)]=id
    };
    println@Console("Created wi "+id+" in RID"+rid)();
    sleep@Time(200)()
}
```

```
15
16 define pop_wi1{
17     if(#wi>0){
18         for(rid1.i=0,rid1.i<(#wi-1),rid1.i++){
19             wi.rid1[rid1.i]=wi.rid1[(rid1.i+1)]
20     };
21     undef(wi.rid1[#wi-1])
22 }}
23
24 define pop_wi2{
25     if(#wi>0){
26         for(rid2.i=0,rid2.i<(#wi-1),rid2.i++){
27             wi.rid2[rid2.i]=wi.rid2[(rid2.i+1)]
28     };
29     undef(wi.rid2[#wi-1])
30 }}
31
32 define pop_wi3{
33     if(#wi>0){
34         for(rid3.i=0,rid3.i<(#wi-1),rid3.i++){
35             wi.rid3[rid3.i]=wi.rid3[(rid3.i+1)]
36     };
37     undef(wi.rid3[#wi-1])
38 }}
39
40 main{
41     {while(true){
42         create_and_allocate_wi
43     }}|{
44     while(true){
45         if(#wi.rid1>0){
46             println@Console("RID1 has taken the wi "+wi.rid1[0])();
47             synchronized(wi_lock){
48                 pop_wi1
49             };
50             sleep@Time(1000)()
51         }
52     }}|{
53     while(true){
54         if(#wi.rid2>0){
55             println@Console("RID2 has taken the wi "+wi.rid2[0])();
56             synchronized(wi_lock){
57                 pop_wi2
58             };
59             sleep@Time(1500)()
60         }
61     }}|{
62     while(true){
63         if(#wi.rid3>0){
64             println@Console("RID3 has taken the wi "+wi.rid3[0])();
65             synchronized(wi_lock){
66                 pop_wi3
```

```
67                  };
68                  sleep@Time(750)()
69              }
70      }}
71  }
```

3.4.5.6 Shortest Queue

The ability to allocate a work item to a selected resource chosen from a group of eligible resources on the basis of having the shortest work queue.

Motivation

Shortest Queue provides a means of allocating work items to resources such that the chosen resource should be able to undertake the work item as soon as possible.

JOLIE **Implementation**

Shortest Queue distribution provides a means of allocating work items to resources with the intention of expediting the throughput of a process instance by ensuring that work items are allocated to the resource that is able to undertake them in the shortest possible timeframe. Typically the shortest timeframe means the resource with the shortest work queue although other interpretations are possible.

An offering achieves full support if it satisfies the description for the pattern.

A simple JOLIE implementation of the *Shortest Queue* pattern is provided by means of a construct in the allocation process which gets the shortest queue corresponding to a specific resource and adds the new work item to it.

JOLIE **code example**

Listing 68: Shortest Queue code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define create_and_allocate_wi{
    getCurrentTimeMillis@Time()(id);
    rid=1;
    for(ci=2,ci<4,ci++){
        if(#wi.("rid"+rid)>#wi.("rid"+(ci))){
            rid=ci}
    };
    synchronized(wi_lock){
        wi.("rid"+rid)[#wi.("rid"+rid)]=id
    };
    println@Console("Created wi "+id+" in RID"+rid)();
    sleep@Time(200)()
}

define pop_wi1{
    if(#wi.rid1>0){
```

```
        for(rid1.i=0,rid1.i<(#wi-1),rid1.i++){
            wi.rid1[rid1.i]=wi.rid1[(rid1.i+1)]
    };
    undef(wi.rid1[#wi-1])
}}

define pop_wi2{
    if(#wi.rid2>0){
        for(rid2.i=0,rid2.i<(#wi-1),rid2.i++){
            wi.rid2[rid2.i]=wi.rid2[(rid2.i+1)]
    };
    undef(wi.rid2[#wi-1])
}}

define pop_wi3{
    if(#wi.rid3>0){
        for(rid3.i=0,rid3.i<(#wi-1),rid3.i++){
            wi.rid3[rid3.i]=wi.rid3[(rid3.i+1)]
    };
    undef(wi.rid3[#wi-1])
}}

main{
    {while(true){
        create_and_allocate_wi
    }}|{
    while(true){
        if(#wi.rid1>0){
            println@Console("RID1 has taken the wi "+wi.rid1[0])();
            pop_wi1;
            sleep@Time(1000)()
        }
    }}|{
    while(true){
        if(#wi.rid2>0){
            println@Console("RID2 has taken the wi "+wi.rid2[0])();
            pop_wi2;
            sleep@Time(1500)()
        }
    }}|{
    while(true){
        if(#wi.rid3>0){
            println@Console("RID3 has taken the wi "+wi.rid3[0])();
            pop_wi3;
            sleep@Time(750)()
        }
    }}
}
```

3.4.5.7 Early Distribution

The ability to allocate a work item to a selected resource chosen from a group of eligible resources on the basis of having the shortest work queue.

Motivation

Early Distribution provides a means of notifying resources of upcoming work items ahead of the time at which they need to be (or can be) executed. This is useful where resources are able to provide some form of forward commitment (or booking) indicating that they will execute and complete a work item at some future time. It also provides a means of optimizing the throughput of a case by ensuring that minimal time is spent waiting for resource allocation during case execution.

JOLIE **Implementation**

Where a process contains a task that is identified as being subject to *Early Distribution*, the existence of any work items corresponding to the task can be advertised to resources as soon as an instance of a process is initiated. Depending on the nature of that implementation, these advertisements may simply be an advance notification or constitute an actual offer/allocation of a work item.

However in both cases, such notifications do not imply that the work item is ready for execution and it is only when the process advances to the task to which the work item corresponds, that the work item can actually be commenced.

An offering achieves full support if it satisfies the description for the pattern.

In most part of the JOLIE implementations provided previously (e.g. the one given for the *Shortest Queue* pattern [3.4.5.6]), each resource has a work item queue related to it, in which each resource takes a work item according to a generic first in, first out (FIFO) policy. Such approach makes the resources themselves (and the work item allocator) aware of information about their future work items and workload.

3.4.5.8 Distribution on Enablement

The ability to advertise and distribute a work items to resources at the moment that the task to which it corresponds is enabled for execution.

Motivation

The simultaneous advertisement and distribution of a work item when the task to which it corresponds is enabled, constitutes the simplest approach to work distribution from a resource perspective as it ensures that any work item that a resource receives in its work list can be immediately acted upon.

JOLIE **Implementation**

Distribution of a work item at the time that the task to which it corresponds is enabled for execution is effectively the standard mechanism for work distribution.

The enablement of a task serves as the trigger for the system to create an associated work item and make it available to resources for execution. This may occur indirectly by placing it on the work lists for individual resources or on the global work list or directly by allocating it to a specific resource for immediate execution.

An offering achieves full support if it satisfies the description for the pattern.

The JOLIE implementation of the *History-based Distribution* pattern [3.4.4.9] is an example of a *Distribution on Enablement* pattern too.

As matter of facts, the distribution of the work item happens with the act of invocation of the resource. In a more general context, the distribution on enablement is the typical (and native) way a work item is distributed to its resource by means of the resource enablement itself.

3.4.5.9 Late Distribution

The ability to advertise and distribute work items to resources after the task to which the work item corresponds has been enabled for execution.

Motivation

Late Distribution of work items effectively provides a means of "demand driving" a process by only advertising or allocating work items to resources after the tasks to which they correspond have already been enabled for execution. This could potentially be much later than the time the tasks were enabled. By adopting this approach, it is possible to reduce the current volume of work in progress within a process instance.

Often this strategy is undertaken with the aim of preventing resources from becoming overwhelmed by the apparent workload even though they may not be required to undertake all of it themselves.

JOLIE **Implementation**

Where a task is identified as being subject to *Late Distribution*, the enablement of the task does not necessarily result in the associated work items being distributed to resources for execution.

Generally other factors are taken into consideration (e.g. number of active work items, available resources, etc.) before the decision is made to advise resources of its existence. This approach to work distribution provides the system with flexibility in determining when work items are made available for execution and offers the potential to reduce context switching when resources have multiple work items that they are attempting to deal with.

This approach to work distribution is often used in conjunction with "heads down" processing where the focus is on maximizing work throughput and the distribution of work is largely under the auspices of the system. At the other end of the spectrum to this approach is the *Case Handling* patter [3.4.4.6] where the distribution and management of work is largely at the discretion of individual resources.

An offering achieves full support if it satisfies the description for the pattern.

The JOLIE implementation of the *Late Distribution* pattern is provided by means of an additional queue, that acts as a unlimited buffer list, in which late-distributed items are stored until the conditions of their execution are met.

JOLIE **code example**

Listing 69: Late Distribution code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define create_and_allocate_wi{
    getCurrentTimeMillis@Time()(id);
    random@Math()(rid);
    rid=1+int(rid*2);
    synchronized(wi_lock){
        if (#wi.("rid"+rid)<5){
            wi.("rid"+rid)[#wi.("rid"+rid)]=id;
            println@Console("Created wi "+id+" in RID"+rid)()
        }
        else{
            wi.("bufferRid"+rid)[#wi.("bufferRid"+rid)]=id;
            println@Console("Created wi "+id+" in buffer RID"+rid)()
        }};
    sleep@Time(500)()
}

define print_stat{
    println@Console("---=== BUFFERS AND QUEUES ===---")();
    println@Console("\tRID1: q:"+#wi.rid1+" b:"+#wi.bufferRid1)();
    println@Console("\tRID2: q:"+#wi.rid2+" b:"+#wi.bufferRid2)();
    println@Console("---============================---")()
}

define pop_wi1{
    if(#wi.rid1>0){
        for(rid1.i=0,rid1.i<(#wi-1),rid1.i++){
            wi.rid1[rid1.i]=wi.rid1[(rid1.i+1)]
    };
    if(#wi.bufferRid1>0){
        wi.rid1[#wi.rid1-1]=#wi.bufferRid1[0]
    }
}}

define pop_wi2{
    if(#wi.rid2>0){
        for(rid2.i=0,rid2.i<(#wi-1),rid2.i++){
            wi.rid2[rid2.i]=wi.rid2[(rid2.i+1)]
    };
    if(#wi.bufferRid2>0){
        wi.rid2[#wi.rid2-1]=#wi.bufferRid2[0]
    }
}}

define pop_buffer_wi1{
    if(#wi.bufferRid1>0){
        for(bufferRid1.i=0,bufferRid1.i<(#wi-1),bufferRid1.i++){
```

```
51              wi.bufferRid1[bufferRid1.i]=wi.bufferRid1[(bufferRid1.i+1)]
52      };
53      undef(wi.bufferRid1[#wi-1])
54  }}
55
56  define pop_buffer_wi2{
57      if(#wi.bufferRid2>0){
58          for(bufferRid2.i=0,bufferRid2.i<(#wi-1),bufferRid2.i++){
59              wi.bufferRid2[bufferRid2.i]=wi.bufferRid2[(bufferRid2.i+1)]
60      };
61      undef(wi.bufferRid2[#wi-1])
62  }}
63
64
65  main{
66      {while(true){
67          create_and_allocate_wi;
68          synchronized(wi_lock){print_stat}
69      }}|{
70      while(true){
71          if(#wi.rid1>0){
72              synchronized(wi_lock){
73                  println@Console("RID1 has taken the wi "+wi.rid1[0])();
74                  pop_wi1;
75                  pop_buffer_wi1
76              };
77              sleep@Time(1000)()
78          }
79      }}|{
80      while(true){
81          if(#wi.rid2>0){
82              synchronized(wi_lock){
83                  println@Console("RID2 has taken the wi "+wi.rid2[0])();
84                  pop_wi2;
85                  pop_buffer_wi2
86              };
87          sleep@Time(1500)()
88      }}}
89  }
```

3.4.6 Pull Patterns

Pull Patterns correspond to the situation where individual resources are made aware of specific work items, that require execution, either via a direct offer from the system or indirectly through a shared work list.

The commitment to undertake a specific task is initiated by the resource itself rather than the system.

Generally this results in the work item being placed on the specific work list for the individual resource for later execution although in some cases, the resource may elect to commence execution on the work item immediately.

The various state transitions associated with pull patterns are illustrated in Figure 3.47.

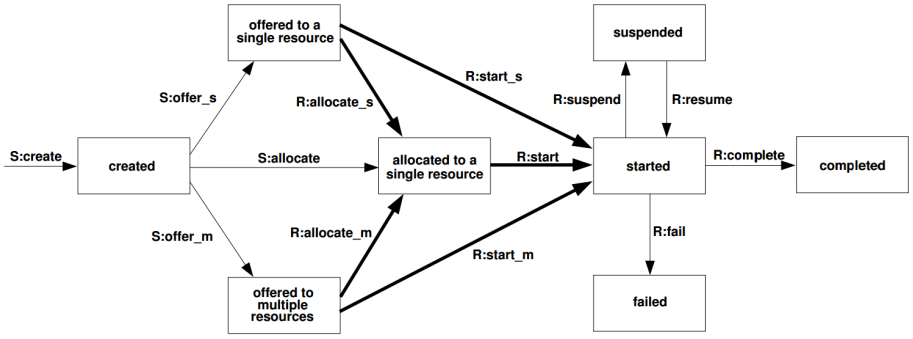


Figure 3.47: Push Patterns

Six pull patterns have been identified. These divide into two distinct groups:

- the first three patterns identify the specifics of the actual "pull" action initiated by the resource, with a particular focus on the work item state before and after the interaction. These patterns correspond to the bold arcs in Figure 3.47;
- the second group of patterns focus on the sequence in which the work items are presented to the resource and the ability of the system and the individual resource to influence the sequence and manner in which they are displayed. The final pattern in this group illustrates the degree of freedom that the resource has in selecting the next work item to execute. These patterns do not

have a direct analogue in Figure 3.47, but apply to all of the "pull" transitions illustrated as bold arcs.

3.4.6.1 Resource-Initiated Allocation

The ability for a resource to commit to undertake a work item without needing to commence working on it immediately.

Motivation

This pattern provides a means for a resource to signal its intention to execute a given work item at some point, although it may not commence working on it immediately.

JOLIE **Implementation**

There are two variants of this pattern as illustrated by the bold arcs in Figure 3.47, depending on whether the work item has been offered to a single resource (*R:allocate_s*) or to multiple resources (*R:allocate_m*).

In both cases, the work item has its status changed from offered to allocated. It remains in the work list of the resource which initiated the allocation.

In the latter case, the work item has been offered to multiple resources and it is therefore necessary to remove it from all other work lists in which it may have appeared as an offer. This ensures that only the resource to which it is now allocated can actually commence working on it.

An offering achieves full support if it satisfies the description for the pattern. It achieves a partial support rating if there are any side effects associated with the implementation of the pattern.

Both the JOLIE implementations of the two variants of the *Resource-Initiated Allocation* pattern (single and multiple) can be seen as a slight modification of the paradigm provided by the two implementations of the *Distribution by Allocation - Single Resource* [3.4.5.3] and *Distribution by Offer - Multiple Resources* [3.4.5.2] patterns.

The modifications of both examples concern the event of work item allocation (either directed or added into a shared work item list). In these cases it's not the system that independently allocates work items to resource(s) (push); instead it's the single resource that notifies the system about its availability to undertake a

work item, which compel the system to allocate a new work item to the resource asking for it (pull).

As it follows, it's provided the JOLIE code example of the single variant of the *Resource-Initiated Allocation* pattern.

JOLIE code example

Listing 70: Resource-Initiated Allocation (single variant) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

inputPort RIA{
    Location:"socket://localhost:8000"
    Protocol:sodep
    OneWay: allocation_request
}

outputPort RIA{
    Location:"socket://localhost:8000"
    Protocol:sodep
    OneWay: allocation_request
}

define pop_rid1{
    if(#wi_stack.rid1>1){
        for(rid1.i=0,rid1.i<#wi_stack.rid1,rid1.i++){
        wi_stack.rid1[rid1.i]=wi_stack.rid1[rid1.i+1]
    }};
    undef(wi_stack.rid1[#wi_stack.rid1-1])
}

define create_and_allocate_wi{
    getCurrentTimeMillis@Time()(id);
    synchronized(wi_lock){
        wi_stack.("rid"+rid)[#wi_stack.("rid"+rid)]=id
    };
    println@Console("Created wi "+id+" as requested from RID"+rid)()
}

main{
    {while(true){
        allocation_request(rid);
        create_and_allocate_wi
    }}|{{
    while(true){
        // WORK ITEM REQUEST
        if(#wi_stack.rid1<5){
            allocation_request@RIA(1)
    }}}|{
    while(true){
```

```
44          //WORK ITEM EXECUTION
45          if(#wi_stack.rid1>0){
46              println@Console("RID1 processes wi "+wi_stack.rid1[0])();
47              synchronized(wi_lock){
48                  pop_rid1
49              };
50              sleep@Time(1000)()
51          }
52      }}}
53  }
```

3.4.6.2 Resource-Initiated Execution - Allocated Work Item

The ability for a resource to commence work on a work item that is allocated to it.

Motivation

Where a resource has work items that it has committed to execute, but has not yet commenced, a means of signaling their commencement is required. This pattern fulfills that requirement.

JOLIE **Implementation**

This pattern corresponds to the *R:start* transition illustrated in Figure 3.47. It results in the status of the selected work item being changed from allocated to started. It remains in the same work list.

The general means of handling that a work item has been allocated to a resource is to place it on a resource-specific work queue. This ensures that the work item is not undertaken by another resource and that the commitment made by the resource to which it is allocated is maintained.

No JOLIE specific implementation of this pattern is provided since, as seen until this point, aside for the *Direct Distribution* [3.4.4.1] and the like (e.g. *History-based Distribution* [3.4.4.9]), the most part of previously analyzed patterns (and their corresponding implementations) defines the allocation of a work item as a queuing operation whose complementary function, put in place by the resource, is the withdrawal of a work item within its own work item list to execute it, which is exactly the behavioral policy described by the *Resource-Initiated Execution - Allocated Work Item* pattern.

3.4.6.3 Resource-Initiated Execution - Offered Work Item

The ability for a resource to select a work item offered to it and commence work on it immediately.

Motivation

In some cases it is preferable to view a resource as being committed to undertaking a work item only when the resource has actually indicated that it is working on it. This approach to work distribution effectively speeds throughput by eliminating the notion of work item allocation. Work items remain on offer to the widest range of appropriate resources until one of them actually indicates they can commence work on it. Only at this time is the work item removed from being on offer and allocated to a specific resource.

JOLIE **Implementation**

There are two variants of this pattern as illustrated by the bold arcs in Figure 3.47, depending on whether the work item has been offered to a single resource (*R:start_s*) or to multiple resources (*R:start_m*). In both cases, the work item has its status changed from offered to started. It remains in the work list of the resource which initiated the work item. In the latter case, the work item has been offered to multiple resources and it is therefore necessary to remove it from all other work lists in which it may have appeared as an offer. This ensures that only one resource can actually work on it.

As stated for the previous *Resource-Initiated Execution - Allocated Work Item* pattern [3.4.6.2], the policy defined by the *Resource-Initiated Execution - Offered Work Item* pattern does not need any further implementation example, since its behaviors (either multiple and single) is exemplified by the *Distribution by Offer - Single Resource* [3.4.5.1] and *Distribution by Offer - Multiple Resources* [3.4.5.2] patterns in which each resource can choose and execute a work item offered into a specific or shared work item list.

3.4.6.4 System-Determined Work Queue Content

The ability of the system to order the content and sequence in which work items are presented to a resource for execution.

Motivation

This pattern provides the system with the ability to specify the ordering and content of work items in a resource's work list. In doing so, the intention is that the system can influence the sequence in which concurrent work items are executed by resources by managing the information presented for each work item.

JOLIE **Implementation**

Where an offering provides facilities for specifying the default ordering in which work items are presented to resources, the opportunity exists to enforce a work ordering policy for all resources or on a group-by-group or individual resource basis. Such ordering may be time-based (e.g. FIFO, LIFO, EDD) or relate to data values associated with individual work items (e.g. cost, required effort, completion time). The ordering and content of work lists can be specified individually for each user or on a whole-of-process basis.

The JOLIE code example provided for this pattern exemplifies processes execution (work item) scheduling. If too much processes are in the execution queue, the system reorders it according to their estimated computation time, this is done to increase the process completion rate and thus lowering the queue size. Once the size of the queue is lowered, the default order, based on processes submission time, is restored. It's worth noting that, to avoid long-computation-time processes starvation, a counter is set to restore the default ordering of one iteration every ten.

JOLIE **code example**

Listing 71: System-Determined Work Queue Content code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define pop_rid1{
    if(#wi_stack.rid1>0){
        for(pop.i=0,pop.i<(#wi_stack.rid1-1),pop.i++){

            wi_stack.rid1[pop.i]<<wi_stack.rid1[(pop.i+1)]
        };
        undef(wi_stack.rid1[#wi_stack.rid1-1])
```

```
    }
}

define order_wi{
    sort=true;
    while(sort){
        sort=false;
        for(obd.i=0,obd.i<#wi_stack.rid1-1,obd.i++){
            if(wi_stack.rid1[obd.i].(ord_arg)>
                wi_stack.rid1[obd.i+1].(ord_arg)){
                temp<<wi_stack.rid1[obd.i];
                wi_stack.rid1[obd.i]<<wi_stack.rid1[obd.i+1];
                wi_stack.rid1[obd.i+1]<<temp;
                sort=true;
                ordered=true
    }}};
    if(ordered){
        println@Console("RID1 wi list reordered by "+ord_arg)()
    }
}

define print_rid1_wi_stack{
    println@Console("---=== RID1 WI STACK ===---")();
    for(pi=0,pi<#wi_stack.rid1,pi++){
        println@Console("id:"+wi_stack.rid1[pi].id+
            " comp_time: "+(wi_stack.rid1[pi].comp_time)+"ms")()
    };
    println@Console("---======================---")()

}

define create_and_allocate_wi{
    random@Math()(wi.comp_time);
    wi.comp_time=int(2000.0*wi.comp_time);
    getCurrentTimeMillis@Time()(wi.id);
    wi_stack.rid1[#wi_stack.rid1]<<wi;
    println@Console("Created wi "+wi.id+
        " with computation time: "+(wi.comp_time/1000)+"s")()
}

main{
    {while(true){
        synchronized(wi_lock){
            create_and_allocate_wi;
            if(#wi_stack.rid1>5 && starv<10){
                ord_arg="comp_time";
                starv++
            }
            else{
                ord_arg="id"
            };
            order_wi;
            print_rid1_wi_stack
        };
```

```
66          sleep@Time(800)()
67      }}|{{
68      while(true){
69          // WORK ITEM REQUEST
70          if(#wi_stack.rid1>0){
71              synchronized(wi_lock){
72                  undef(rid1.wi);
73                  rid1.wi<<wi_stack.rid1[0];
74                  println@Console("RID1 processes wi "+
75                      rid1.wi.id)();
76                  pop_rid1;
77                  starv=0
78              };
79              sleep@Time(rid1.wi.comp_time)()
80          }
81      }}}
82  }
```

3.4.6.5 Resource-Determined Work Queue Content

The ability for resources to specify the format and content of work items listed in the work queue for execution.

Motivation

Enabling resources to specify the format, content and ordering of their work queue provides them with a greater degree of flexibility in both the selection of offered work items for execution and also in how they tackle work items which they have committed to execute or have been allocated to them.

JOLIE **Implementation**

Typically this pattern manifests itself as the availability of a range of sorting and filtering options that resources can access to tailor the format of their work list. These options may be either transient views that they can request or alternately can take the form of permanent configuration options for their work lists.

The JOLIE code example provided for this pattern can be taken as a slight modification of the one given for the previous *System-Determined Work Queue Content* pattern [3.4.6.4], where in this case is the resource the one who chooses what is the best sorting policy for its own work items.

JOLIE **code example**

Listing 72: Resource-Determined Work Queue Content code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define pop_rid1{
    if(#wi_stack.rid1>0){
        for(pop.i=0,pop.i<(#wi_stack.rid1-1),pop.i++){

            wi_stack.rid1[pop.i]<<wi_stack.rid1[(pop.i+1)]
        };
        undef(wi_stack.rid1[#wi_stack.rid1-1])
    }
}

define order_wi{
    sort=true;
    while(sort){
        sort=false;
        for(obd.i=0,obd.i<#wi_stack.rid1-1,obd.i++){
            if(wi_stack.rid1[obd.i].(ord_arg)>
```

```
                wi_stack.rid1[obd.i+1].(ord_arg)){
                temp<<wi_stack.rid1[obd.i];
                wi_stack.rid1[obd.i]<<wi_stack.rid1[obd.i+1];
                wi_stack.rid1[obd.i+1]<<temp;
                sort=true;
                ordered=true
    }}};
    if(ordered){
        println@Console("RID1 wi list reordered by "+ord_arg)()
    }
}

define print_rid1_wi_stack{
    println@Console("---=== RID1 WI STACK ===---")();
    for(pi=0,pi<#wi_stack.rid1,pi++){
        println@Console("id:"+wi_stack.rid1[pi].id+
            " comp_time: "+(wi_stack.rid1[pi].comp_time)+"ms")()
    };
    println@Console("---======================---")()

}

define create_and_allocate_wi{
    random@Math()(wi.comp_time);
    wi.comp_time=int(2000.0*wi.comp_time);
    getCurrentTimeMillis@Time()(wi.id);
    wi_stack.rid1[#wi_stack.rid1]<<wi;
    println@Console("Created wi "+wi.id+
        " with computation time: "+(wi.comp_time/1000)+"s")()
}

main{
    {while(true){
        synchronized(wi_lock){
        create_and_allocate_wi};
        sleep@Time(800)()
    }}|{
    while(true){
        // WORK ITEM REQUEST
        if(#wi_stack.rid1>0){
            if(#wi_stack.rid1>5 && starv<10){
                ord_arg="comp_time";
                starv++
            }
            else{
                ord_arg="id"
            };
            order_wi;
            print_rid1_wi_stack;
            synchronized(wi_lock){
                undef(rid1.wi);
                rid1.wi<<wi_stack.rid1[0];
                println@Console("RID1 processes wi "+
```

```
74                      rid1.wi.id)();
75                  pop_rid1;
76                  starv=0
77              };
78              sleep@Time(rid1.wi.comp_time)()
79      }}}
80  }
```

3.4.6.6 Selection Autonomy

The ability for resources to select a work item for execution based on its characteristics and their own preferences.

Motivation

The ability for a resource to select the work item that they will commence next is a key aspect of the "heads up" approach to workflow execution. It aims to empower resources and let them have the flexibility to prioritize and organize their own individual work sequence.

JOLIE **Implementation**

This pattern is a common feature and it typically manifests itself in one of two forms:

- a resource is able to execute multiple work items simultaneously and thus can initiate additional work items of their choice at any time;
- resources are limited to executing one work item at a time, in which case they can only commence a new work item when the previous one is complete, although they can choose which work item they will commence next.

Where a system implements "heads down" processing, it is common for the *Selection Autonomy* pattern to be disabled and for the system to determine which work item a resource will execute next.

The JOLIE code example provided for this pattern can be based upon the one given for the *System-Determined Work Queue Content* pattern [3.4.6.4], where in this case three resources can pick their preferred work items (sequentially), each of them applying different policies: RID1 chooses randomly among the whole list of work items, while RID2 and RID3 implement a deterministic behavior, RID2 always choosing the last (computationally longest) work item and RID3 that always chooses the first (computationally shortest) work item of the list.

JOLIE **code example**

Listing 73: Selection Autonomy code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define pop_wi{
    if(#wi_stack>0){
        for(pop.i=pop_wi_n,pop.i<(#wi_stack-1),pop.i++){
            wi_stack[pop.i]<<wi_stack[(pop.i+1)]
    };
    undef(wi_stack[#wi_stack-1])
}}

define order_wi{
    sort=true;
    while(sort){
        sort=false;
        for(obd.i=0,obd.i<#wi_stack-1,obd.i++){
            if(wi_stack[obd.i].comp_time>
                wi_stack[obd.i+1].comp_time){
                temp<<wi_stack[obd.i];
                wi_stack[obd.i]<<wi_stack[obd.i+1];
                wi_stack[obd.i+1]<<temp;
                sort=true
    }}}
}

define print_wi_stack{
    println@Console("---=== WI STACK ===---")();
    for(pi=0,pi<#wi_stack,pi++){
        println@Console("id:"+wi_stack[pi].id+
            " comp_time: "+(wi_stack[pi].comp_time)+"ms")()
    };
    println@Console("---================---")()

}

define create_and_allocate_wi{
    random@Math()(wi.comp_time);
    wi.comp_time=int(3000.0*wi.comp_time);
    getCurrentTimeMillis@Time()(wi.id);
    wi_stack[#wi_stack]<<wi;
    println@Console("Created wi "+wi.id+
        " with computation time: "+wi.comp_time+"ms")()
}

main{
    {while(true){
        synchronized(wi_lock){
            create_and_allocate_wi;
            order_wi;
```

```
51              print_wi_stack
52          };
53          sleep@Time(500)()
54      }}|{
55      while(true){
56          synchronized(wi_lock){
57              if(#wi_stack>0){
58                  random@Math()(wid);
59                  wid=int(wid*#wi_stack);
60                  println@Console("RID1 has taken the wi "+
61                      wi_stack[wid].id)();
62                  rid1.comp_time=wi_stack[wid].comp_time;
63                  pop_wi_n=wid;
64                  pop_wi
65              }};
66          sleep@Time(rid1.comp_time)()
67      }}|{
68      while(true){
69          synchronized(wi_lock){
70              if(#wi_stack>0){
71                  println@Console("RID2 has taken the wi "+
72                      wi_stack[#wi_stack-1].id)();
73                  rid2.comp_time=wi_stack[#wi_stack-1].comp_time;
74                  undef(wi_stack[#wi_stack-1])
75              }};
76          sleep@Time(rid2.comp_time)()
77      }}|{
78      while(true){
79          synchronized(wi_lock){
80              if(#wi_stack>0){
81                  println@Console("RID3 has taken the wi "+
82                      wi_stack[0].id)();
83                  rid3.comp_time=wi_stack[0].comp_time;
84                  pop_wi_n=0;
85                  pop_wi
86              }};
87          sleep@Time(rid3.comp_time)()
88      }}
89  }
```

3.4.7 Detour Patterns

Detour Patterns refer to situations where work item distributions that have been made for resources are interrupted either by the system or at the instigation of the resource.

As a consequence of this event, the normal sequence of state transitions for a work item is varied. The range of possible scenarios for detour patterns are illustrated in Figure 3.48

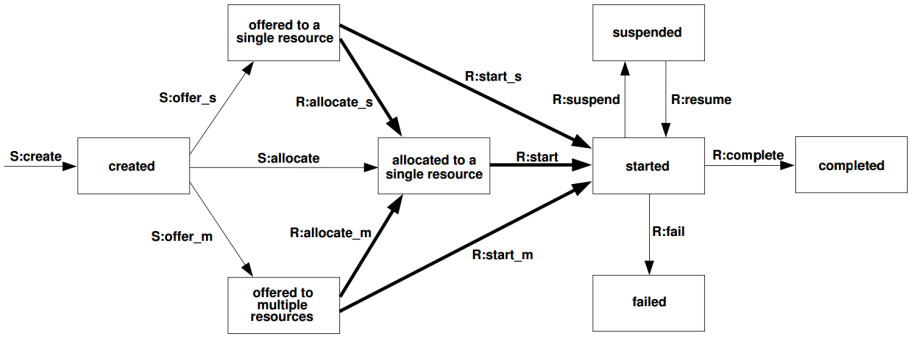


Figure 3.48: Detour Patterns

There are a number of possible impacts on a work item, depending on its current state of progression and whether the detour was initiated by the resource with which the work item was associated or by the system. A brief definition of these patterns has been given during the analysis of the possible user privileges on tasks, in the *Authorization* pattern [3.4.4.4] description.

3.4.7.1 Delegation

The ability for a resource to allocate a not-started work item previously allocated to it (but not yet commenced) to another resource.

Motivation

Delegation provides a resource with a means of re-routing work items that it is unable to execute. This may be because the resource is going to be unavailable or because it does not wish to take on any more work.

JOLIE Implementation

Delegation is usually initiated by a resource via their work list handler. It removes a work item that is allocated to them (but not yet commenced) and inserts it into the work list of another nominated resource. It is illustrated by the *R:delegate* transition in Figure 3.48.

An offering achieves full support if it satisfies the description for the pattern.

The JOLIE implementation of this pattern has been already used in other patterns implementations, e.g. the *Distribution by Offer - Single Resource* pattern [3.4.5.1] where a work item is offered to a specific resource in a non-binding basis, such that the interested resource can choose either to process the offered work item or to offer it to another resource, by adding it to its work items list.

For the sake of brevity, snippet of code provided as it follows refers only to the implementation of the *Delegation* pattern present in the *Distribution by Offer - Single Resource* code example. In this particular example the resource RID1 can choose (on random basis) whether to process a work item offered to it or to delegate it to the resource RID2 .

JOLIE code example

Listing 74: Delegation code (snippet) example

```
...
if(#wi_stack.rid1>0){
    random@Math()(rid1.deleg);
    rid1.deleg=int(rid1.deleg*2);
    if(rid1.deleg){
        synchronized(wi_lock){
            wi_stack.rid2[#wi_stack.rid2]=wi_stack.rid1[0];
            println@Console("RID1 offered the wi "+wi_stack.rid1[0]+
                " to RID2")();
            sleep@Time(500)()
        }
...
```

3.4.7.2 Escalation

The ability of a system to distribute a work item to a resource or group of resources other than those it has previously been distributed to in an attempt to expedite the completion of the work item.

Motivation

Escalation provides the ability for a system to intervene in the conduct of a work item and assign it to alternative resources. Generally this occurs as a result of a specified deadline being exceeded, but it may also be a consequence of preemptive load balancing of work allocations undertaken by the system or manually by the process administrator in an attempt to optimize workflow throughput.

JOLIE **Implementation**

There are various ways in which a work item may be escalated depending on its current state of progression and the approach that is taken to identifying a suitable party to which it should be reassigned.

The possible range of alternatives are illustrated by the *S:escalate_(argument)* transitions in Figure 3.48.

An escalation action is triggered by the system or process administrator and results in the work item being removed from the work lists of all resources to which it was previously offered or allocated and added to the work lists of the users to which it is being reassigned in either an offered or allocated state.

An offering achieves full support if it satisfies the description for the pattern.

The JOLIE implementation of this pattern is provided by means of a slight modification of the example given for the *Selection Autonomy* pattern [3.4.6.6] in which, once the queue of work items assigned to RID1 and RID2 becomes longer than a a guard value, the exceeding items in the queue are escalated (reassigned) into the work items list of the resource RID3.

JOLIE **code example**

Listing 75: Escalation code example

```
1 include "console.iol"
2 include "time.iol"
3 include "math.iol"
4
```

```
define pop_wi{
    if(#wi_stack.main>0){
        for(pop.i=pop_wi_n,pop.i<(#wi_stack.main-1),pop.i++){
            wi_stack.main[pop.i]<<wi_stack.main[(pop.i+1)]
    };
    undef(wi_stack.main[#wi_stack.main-1])
}}

define pop_wi_rid3{
    if(#wi_stack.rid3>0){
        for(pop.i=0,pop.i<(#wi_stack.rid3-1),pop.i++){
            wi_stack.rid3[pop.i]<<wi_stack.rid3[(pop.i+1)]
    };
    undef(wi_stack.rid3[#wi_stack.rid3-1])
}}

define order_wi{
    sort=true;
    while(sort){
        sort=false;
        for(obd.i=0,obd.i<#wi_stack.main-1,obd.i++){
            if(wi_stack.main[obd.i].comp_time>
                wi_stack.main[obd.i+1].comp_time){
                temp<<wi_stack.main[obd.i];
                wi_stack.main[obd.i]<<wi_stack.main[obd.i+1];
                wi_stack.main[obd.i+1]<<temp;
                sort=true
    }}}
}


define print_wi_stack{
    println@Console("---=== WI STACK ===---")();
    for(pi=0,pi<#wi_stack.main,pi++){
        println@Console("id:"+wi_stack.main[pi].id+
            " comp_time: "+(wi_stack.main[pi].comp_time)+"ms")()
    };
    println@Console("---================---")()

}

define print_wi_stack_rid3{
    println@Console("---=== RID3 WI STACK ===---")();
    for(pi=0,pi<#wi_stack.rid3,pi++){
        println@Console("id:"+wi_stack.rid3[pi].id+
            " comp_time: "+(wi_stack.rid3[pi].comp_time)+"ms")()
    };
    println@Console("---================---")()

}

define escalate_to_rid3{
    if(#wi_stack.rid3<#wi_stack.main){
        wi_stack.rid3[#wi_stack.rid3]<<wi_stack.main[ei];
        pop_wi_n=0;
```

```
60          pop_wi;
61          print_wi_stack_rid3
62      }
63  }
64
65  define create_and_allocate_wi{
66      random@Math()(wi.comp_time);
67      wi.comp_time=int(3000.0*wi.comp_time);
68      getCurrentTimeMillis@Time()(wi.id);
69      wi_stack.main[#wi_stack.main]<<wi;
70      println@Console("Created wi "+wi.id+
71          " with computation time: "+wi.comp_time+"ms")()
72  }
73
74  main{
75      {while(true){
76          synchronized(wi_lock){
77              create_and_allocate_wi;
78              order_wi;
79              if(#wi_stack.main>5){
80                  escalate_to_rid3
81              };
82              print_wi_stack
83          };
84          sleep@Time(500)()
85      }}|{
86      while(true){
87          synchronized(wi_lock){
88              if(#wi_stack.main>0){
89                  random@Math()(wid);
90                  wid=int(wid*#wi_stack.main);
91                  println@Console("RID1 has taken the wi "+
92                      wi_stack.main[wid].id)();
93                  rid1.comp_time=wi_stack.main[wid].comp_time;
94                  pop_wi_n=wid;
95                  pop_wi
96              }};
97          sleep@Time(rid1.comp_time)()
98      }}|{
99      while(true){
100         synchronized(wi_lock){
101             if(#wi_stack.main>0){
102                 println@Console("RID2 has taken the wi "+
103                     wi_stack.main[#wi_stack.main-1].id)();
104                 rid2.comp_time=wi_stack.main[#wi_stack.main-1].comp_time;
105                 undef(wi_stack.main[#wi_stack.main-1])
106             }};
107         sleep@Time(rid2.comp_time)()
108     }}|{
109     while(true){
110         synchronized(wi_lock){
111             if(#wi_stack.rid3>0){
```

```
112                 println@Console("RID3 has taken the wi "+
113                     wi_stack.rid3[0].id)();
114                 rid3.comp_time=wi_stack.rid3[0].comp_time;
115                 pop_wi_rid3
116             }};
117         sleep@Time(rid3.comp_time)()
118     }}
119 }
```

3.4.7.3 Deallocation

The ability of a resource (or group of resources) to relinquish a work item which is allocated to it (but not yet commenced) and make it available for distribution to another resource or group of resources.

Motivation

Deallocation provides resources with a means of relinquishing work items allocated to them and making them available for re-distribution to other resources. This may occur for a variety of reasons including insufficient progress, availability of a better resource or a general need to unload work from a resource.

JOLIE **Implementation**

There are two possible variations to *Deallocation* either the work item can be offered to a single resource or to multiple resources.

An offering achieves full support if it satisfies the description for the pattern.

The JOLIE code example provided for previous patterns, e.g. *Escalation* [3.4.7.2], necessitated the implicit implementation of the *Deallocation* pattern since, while work items are escalated from a list to another, each of these work items is deallocated from the *main* list (the one shared between resources RID1 and RID2) and consequently reallocated (see further) in the RID3 queue.

For the sake of brevity, only the snippet of code containing the operation of deallocation is reported as it follows.

JOLIE **code example**

Listing 76: Deallocation code (snippet) example

```
...
    if(#wi_stack.rid3<#wi_stack.main){
        wi_stack.rid3[#wi_stack.rid3]<<wi_stack.main[ei];
        pop_wi_n=0;
        pop_wi;
        print_wi_stack_rid3
    }
...
```

3.4.7.4 Stateful Reallocation

The ability of a resource to allocate a work item that they are currently executing to another resource without loss of state data.

Motivation

Stateful Reallocation provides a resource with the ability to offload currently executing work items to other resources whilst maintaining the current state of the work item and the results of work undertaken on it to date. In the main, this focuses on the ability to retain the current values of all data elements associated with the work item. It is motivated by the need for a resource to pass on a work item to another resource without losing the benefit of any work that has already been undertaken in regard to it.

JOLIE **Implementation**

Planned reallocation provides a resource with the ability to offload both pending and currently executing work items to other resources whilst maintaining the current state of the work item and the results of work undertaken on it to date.

In the main, this centers on the ability to retain the current values of all data elements associated with the work item. This pattern corresponds to the *R:reallocation* with state arc in Figure 3.48.

It is interesting to note the similarities between this pattern and the *Delegation* pattern [3.4.7.1]. Both patterns result in a work item being reassigned to another resource. The main difference is that *Delegation* can only occur for a work item that has not yet commenced execution where as this pattern applies to work items that are currently being executed.

An offering achieves full support if it satisfies the description for the pattern.

The JOLIE code example given for the implementation of this pattern is based on a "parallel subordinated collaboration" (delegation/escalation) between two resources RID1 and RID2.

Let's see RID1 like an higher level employee (senior) which has to undertake several work items, whose completion requires passing (and working on) three steps. The first step is a general overview of the work that must be done on that work item, and thus only RID1 can undertake it, because of its leading position.

After RID1 has done with the first step of a work item, it can choose either to undertake each of the remaining steps of that work item, or to reallocate it to RID2

(one of its subordinates) which can complete the work on that item from the step RID1 left (reallocated) it; this feature makes sure that the event of reallocating the work item preserves its state. While RID2 is completing the work on the reallocated item, RID1 can continue working on other work items, still been able to reallocate new work items to RID2, after the completion of the first step of that work item.

JOLIE **code example**

Listing 77: Stateful Reallocation code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define pop_wi_rid1{
    if(#wi_stack>0){
        for(pop.i=0,pop.i<(#wi_stack.rid1-1),pop.i++){
            wi_stack.rid1[pop.i]<<wi_stack.rid1[(pop.i+1)]
    };
    undef(wi_stack.rid1[#wi_stack.rid1-1])
}}

define pop_wi_rid2{
    if(#wi_stack>0){
        for(pop.i=0,pop.i<(#wi_stack.rid2-1),pop.i++){
            wi_stack.rid2[pop.i]<<wi_stack.rid2[(pop.i+1)]
    };
    undef(wi_stack.rid2[#wi_stack.rid2-1])
}}

define print_wi_stack{
    for(rids=1,rids<3,rids++){
        println@Console("---=== WI STACK RID"+rids+"===---")();
        for(pi=0,pi<#wi_stack.("rid"+rids),pi++){
            println@Console("id: "+wi_stack.("rid"+rids)[pi].id)();
            for(ri=1,ri<4,ri++){
                print@Console("\tstep"+ri+" comp_time: "+
                    wi_stack.("rid"+rids)[pi].("step"+ri).comp_time
                    +"ms")();
                    if(wi_stack.("rid"+rids)[pi].("step"+ri).done){
                        println@Console(", done")()
                    }else{println@Console()()}
            }
        };
        println@Console("---=================---")()
    }
}

define create_and_offer_wi{
    getCurrentTimeMillis@Time()(wi.id);
```

```
    random@Math()(wi.step1.comp_time);
    wi.step1.comp_time=int(wi.step1.comp_time*500);
    random@Math()(wi.step2.comp_time);
    wi.step2.comp_time=int(wi.step2.comp_time*500);
    random@Math()(wi.step3.comp_time);
    wi.step3.comp_time=int(wi.step3.comp_time*500);
    wi_stack.rid1[#wi_stack.rid1]<<wi;
    println@Console("Created wi "+wi.id)()
}

define stateful_reallocate_to_rid2 {
    if(#wi_stack.rid1>3){
        println@Console("Reallocated to RID2 wi: "+
            rid1.wi.id)();
        synchronized(wi2_lock){wi_stack.rid2<<rid1.wi};
        rid1.sr=true
    }
}

main{
    {while(true){
        synchronized(wi_lock){
            create_and_offer_wi;
            synchronized(wi2_lock){
                print_wi_stack
            }
        };
        sleep@Time(500)()
    }}|{
    while(true){
        rid1.sr=false;
        synchronized(wi_lock){
            if(#wi_stack.rid1>0){rid1.wi->wi_stack.rid1[0]}};
        if(is_defined(rid1.wi)){
            sleep@Time(rid1.wi.step1.comp_time)();
            rid1.wi.step1.done=true;
            for(rid1.step=2,rid1.step<4 && !rid1.sr,rid1.step++){
                synchronized(wi_lock){stateful_reallocate_to_rid2};
                if(!rid1.sr){
                    sleep@Time(rid1.wi.("step"+rid1.step).comp_time)();
                    rid1.wi.("step"+rid1.step).done=true
                }};
            println@Console("RID1 finished processing wi: "+
                rid1.wi.id)();
            pop_wi_rid1;
            undef(rid1.wi)
    }}}|{
    while(true){
        synchronized(wi2_lock){
            if(#wi_stack.rid2>0){rid2.wi->wi_stack.rid2[0]}};
            if(is_defined(rid2.wi)){
                for(rid2.step=2,rid2.step<4,rid2.step++){
```

```
93                      if(!rid2.wi.("step"+rid2.step).done){
94                          sleep@Time(rid2.wi.("step"+
95                              rid2.step).comp_time)();
96                          rid2.wi.("step"+rid2.step).done=true
97                  }};
98                  println@Console("RID2 finished processing wi: "+
99                  rid2.wi.id)();
100                 pop_wi_rid2;
101                 undef(rid2.wi)
102     }}}
103 }
```

3.4.7.5 Stateless Reallocation

The ability for a resource to reallocate a work item that it is currently executing to another resource without retention of state.

Motivation

Stateless Reallocation provides a lightweight means of reallocating a work item to another resource without needing to consider the complexities of state preservation. In effect, when this type of reallocation occurs all state information associated with the work item (and hence any record of effective progress) is lost and the work item is basically restarted by the resource to which it is reassigned.

JOLIE **Implementation**

This pattern is illustrated by the *R:reallocation_no_state* arc in Figure 3.48. It has similarities in terms of outcome with *Delegation* [3.4.7.1] and *Escalation* [3.4.7.2] patterns in that the work item is restarted except that in this scenario, the work item has already been partially executed prior to the restart. This pattern can only be implemented for work items that are capable of being redone without any consequences relating to the previous execution instance(s).

An offering achieves full support if it satisfies the description for the pattern.

Although easier examples of possible implementations in JOLIE of the *Stateless Reallocation* patter can be given, it's worth considering the relation between this pattern and it's stateful correspondent [3.4.7.4].

As stated previously, this pattern can be implemented for work items that are capable of being redone without any consequences. Thus, w.r.t. the kind of work items defined in *Stateful Reallocation* example, the Stateless implementation of the same context example, consists in modifying the reallocating operation, which, prior to reallocate the work item, resets all of the steps done by the resource.

Since the most part of the code example of this implementation is shared with the one given for the *Stateful Reallocation* example, is made reference to it, stateless reallocation operation apart, whose implementation is reported as it follows.

JOLIE **code example**

Listing 78: System-Determined Work Queue Content code example

```
define stateless_reallocate_to_rid2 {
    if(#wi_stack.rid1>3){
        println@Console("Reallocated to RID2 wi: "+
            rid1.wi.id)();
        rid1.wi.step1.done=false;
        rid1.wi.step2.done=false;
        rid1.wi.step3.done=false;
        synchronized(wi2_lock){wi_stack.rid2<<rid1.wi};
        rid1.sr=true
    }
}
```

3.4.7.6 Suspension-Resumption

The ability for a resource to suspend and resume execution of a work item.

Motivation

In some situations, during the course of executing a work item, a resource reaches a point where it is not possible to progress it any further. *Suspension* provides the ability for the resource to signal a temporary halt to the system of any work on the particular work item and switch its attention to another.

JOLIE **Implementation**

Suspension and *Resumption* actions are generally initiated by a resource from their work list handler. A suspended work item remains in the resource's work list but its state is generally notated as suspended. It is able to be restarted at some future time.

This pattern is illustrated by the *R:suspend* and *R:resume* arcs in Figure 3.48.

The use of JOLIE *synchronization* construct is very suitable for the implementation of this pattern, since, "tokenizing" steps execution, the system can "stop" the resource while working on a work item completion, simply by "keeping" the token of the execution synchronization for a certain time (or even waiting for an external input and the like), after which the token is released and the resource can resume the execution.

An offering achieves full support if it satisfies the description for the pattern.

Using the JOLIE *synchronization* construct concerns work item state consistency too, because suspending a running execution might lead to data loss or inconsistency. Contrariwise, if the execution of a step is completing and the *synchronization* construct is used to "atomize" the step sequence, the system has to wait for that step to complete its execution, which at last will release the synchronization token. Then the system can take that token and keep it for the whole time of the execution suspension.

JOLIE **code example**

Listing 79: Suspension-Resumption code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define pop_wi{
    if(#wi_stack>0){
        for(pop.i=0,pop.i<(#wi_stack-1),pop.i++){
            wi_stack[pop.i]<<wi_stack[(pop.i+1)]
    };
    undef(wi_stack[#wi_stack-1])
}}

define print_wi_stack{
    println@Console("---=== WI STACK ===---")();
    for(pi=0,pi<#wi_stack,pi++){
        println@Console("id: "+wi_stack[pi].id)();
        for(ri=0,ri<5,ri++){
            print@Console("\tstep "+ri)();
                if(wi_stack[pi].("step"+ri).done){
                    println@Console(", done")()
                }else{println@Console()()}
        }
    };
    println@Console("---================---")()
}

define create_and_offer_wi{
    getCurrentTimeMillis@Time()(wi.id);
    wi_stack[#wi_stack]<<wi;
    println@Console("Created wi "+wi.id)()
}

main{
    {while(true){
        synchronized(wi_lock){
            synchronized(exec_lock){
                create_and_offer_wi;
                print_wi_stack
            }
        };
        sleep@Time(2500)()
    }}|{while(true){
        sleep@Time(7000)();
        synchronized(exec_lock){
            println@Console("System suspends execution for 3s.")();
            sleep@Time(3000)();
            println@Console("System resumes execution.")()
        }
    }}|{
```

```
50      while(true){
51          synchronized(wi_lock){
52              if(#wi_stack>0){rid1.wi->wi_stack[0]}};
53          if(is_defined(rid1.wi)){
54              for(step=0,step<5,step++){
55              synchronized(exec_lock){
56                  sleep@Time(500)();
57                  rid1.wi.("step"+step).done=true;
58                  println@Console("RID1 completed step "+step+
59                          " of wi: "+rid1.wi.id)()
60              }
61          };
62          synchronized(exec_lock){
63              println@Console("RID1 finished processing wi: "+
64                  rid1.wi.id)();
65              pop_wi;
66              undef(rid1.wi)
67          }
68      }}}
69  }
```

3.4.7.7 Skip

The ability for a resource to skip a work item allocated to it and mark the work item as complete.

Motivation

The ability to skip a work item reflects the common approach to expediting process instances by simply ignoring non-critical activities and assuming them to be complete such that work items associated with subsequent tasks can be commenced.

JOLIE **Implementation**

The *Skip* pattern is generally implemented by providing a means for a resource to advance the state of a work item from allocated to completed. This pattern is illustrated by the *R:skip* arc in Figure 3.4.7.7.

An offering achieves full support if it satisfies the description for the pattern.

The JOLIE code example provided for this pattern is based an a skipping policy set upon the work item queue size. If too many work items are queued and wait for execution, the two last steps of completion are marked as "skip-able", skipped and flagged as completed. Thus, skipping some of the last steps, the resource can shorten the queue of work items waiting for execution.

JOLIE **code example**

Listing 80: Skip code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define pop_wi{
    if(#wi_stack>0){
        for(pop.i=0,pop.i<(#wi_stack-1),pop.i++){
            undef(wi_stack[pop.i]);
            wi_stack[pop.i]<<wi_stack[(pop.i+1)]
    };
    undef(wi_stack[#wi_stack-1])
}}

define print_wi_stack{
    println@Console("---=== WI STACK ===---")();
    for(pi=0,pi<#wi_stack,pi++){
        println@Console("id: "+wi_stack[pi].id)();
```

```
        for(ri=0,ri<5,ri++){
            print@Console("\tstep "+ri)();
                if(wi_stack[pi].("step"+ri).done){
                    println@Console(", done")()
                }else{println@Console()()}
        }
    };
    println@Console("---================---")()
}

define create_and_offer_wi{
    getCurrentTimeMillis@Time()(wi.id);
    wi_stack[#wi_stack]<<wi;
    println@Console("Created wi "+wi.id)()
}

main{
    {while(true){
        synchronized(wi_lock){
            create_and_offer_wi;
            print_wi_stack
        };
        sleep@Time(1500)()
    }}|{
    while(true){
        synchronized(wi_lock){
            if(#wi_stack>0){rid1.wi->wi_stack[0]}};
        if(is_defined(rid1.wi)){
            for(step=0,step<5,step++){
                if(step<3 || #wi_stack<2){
                    sleep@Time(500)()
                }else{
                    synchronized(wi_lock){
                        println@Console(
                            "Too many WIs, skipping step "+    step)()}
                };
                synchronized(wi_lock){
                    rid1.wi.("step"+step).done=true;
                    println@Console("RID1 completed step "+step+
                        " of wi: "+rid1.wi.id)()
                }
        };
        synchronized(wi_lock){
            println@Console("RID1 finished processing wi: "+
            rid1.wi.id)();
            pop_wi;
            undef(rid1.wi)
        }}
    }}
}
```

3.4.7.8 Redo

The ability for a resource to redo a work item that has previously been completed in a case. Any subsequent work items (i.e. work items that correspond to subsequent tasks in the process) must also be repeated.

Motivation

The *Redo* pattern allows a resource to repeat a work item that has previously been completed. This may be based on a decision that the work item was not undertaken properly or because more information has become available that alters the potential outcome of the work item.

JOLIE **Implementation**

The *Redo* pattern effectively provides a means of "winding back" the progress of a case to an earlier task. Some difficulties are associated with doing this, in particular where a process instance involves multiple users, however for situations where all of the work items in a case are allocated to the same user (e.g. in a case handling system), the problem is more tractable.

One consideration in using this pattern is that, whilst it is possible to regress the execution state in a case, it is generally not possible to wind back the state of data elements, hence any necessary reversion of data values needs to be managed at the level of specific applications. This pattern is illustrated by the *R:redo* arc in Figure 3.48.

There is one context condition associated with this pattern: any shared data elements (i.e. block, scope, case data etc.) cannot be destroyed during the execution of a case.

Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption.

The simple, but yet meaningful, *Redo* pattern realization in JOLIE provided as it follows, takes into account the case of an external request, simulated by a waiting operation which fires a *Redo* instruction each 3,5 seconds, that makes the step of execution wind back to the desired (random) index and, updates (re-does) the data of each work item until the end of the queue.

JOLIE **code example**

Listing 81: Redo code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define print_wi_stack{
    println@Console("---=== WI STACK ===---")();
    for(pi=0,pi<#wi_stack,pi++){
        print@Console("id: "+wi_stack[pi].id)();
        if(wi_stack[pi].done){
            println@Console(", done at "+wi_stack[pi].step_comp+
                " step")()
            }
            else{println@Console("")()}
    };
    println@Console("---================---")()
}

define create_and_offer_wi{
    getCurrentTimeMillis@Time()(wi.id);
    wi_stack[#wi_stack]<<wi;
    println@Console("Created wi "+wi.id)()
}

main{
    {while(true){
        synchronized(wi_lock){
            create_and_offer_wi;
            print_wi_stack
        };
        sleep@Time(1500)()
    }}|{while(true){
        sleep@Time(3500)();
        random@Math()(redo_from_step);
        redo_from_step=int(redo_from_step*#wi_stack);
        synchronized(wi_lock){
            rid1.redo_step=redo_from_step;
            println@Console("Redo from step "+redo_from_step)()
        }
    }}|{
    rid1.step=0;
    rid1.redo_step=0;
    rid1.wi->wi_stack[rid1.redo_step];
    while(true){
        synchronized(wi_lock){
            if(#wi_stack>rid1.redo_step){
                sleep@Time(500)();
                rid1.wi.done=true;
                rid1.wi.step_comp=rid1.step;
```

```
                println@Console("RID1 completed wi: "+
                    rid1.wi.id+" at step "+rid1.wi.step_comp)();
                rid1.step++;
                rid1.redo_step++
            }
        }}}
}
```

3.4.7.9 Pre-Do

The ability for a resource to execute a work item ahead of the time that it has been offered or allocated to resources working on a given case. Only work items that do not depend on data elements from preceding work items can be "pre-done".

Motivation

The *Pre-Do* pattern provides resources with the ability to complete work items in a case ahead of the time that they are required to be executed i.e. prior to them being offered or allocated to resources working on the case. The motivation for this being that overall throughput of the case may be expedited by completing work items as soon as possible regardless of the order in which they appear in the actual process specification.

JOLIE **Implementation**

The *Pre-Do* pattern effectively provides a means of completing the work items in a case in a user-selected sequence. There are difficulties associated with doing this where later work items rely on data elements from earlier work items.

However for situations where all of the work items in a case are allocated to the same user and there is less data coupling or the implications of shared data can be managed by resources (e.g. in a case handling system), the problem is more tractable. This pattern is not illustrated in Figure 3.48.

There is one context condition associated with this pattern: any shared data elements (i.e. block, scope, case data etc.) must be created at the beginning of the case.

Full support for this pattern is demonstrated by any offering which provides a construct which satisfies the description when used in a context satisfying the context assumption.

The JOLIE code example for this pattern brings the situation in which a resource (RID1) is composed by two “parts” that run in parallel:

- one, that’s the more common resource implementation as seen before, that picks the first available (not taken, nor done) work item and executes it, checking if its dependencies are met;
- the other, which implements the *Pre-do* behavior, picks the first available (not taken, nor done) work item that has no dependencies and executes it.

JOLIE **code example**

Listing 82: Pre-Do code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define print_wi_stack{
    println@Console("---=== WI STACK ===---")();
    for(pi=0,pi<#wi_stack,pi++){
        print@Console("id: "+wi_stack[pi].id)();
        if(is_defined(wi_stack[pi].depends)){
            print@Console(", depends from wi: "+
                wi_stack[pi].depends)()
        };
        if(wi_stack[pi].done){
            println@Console(", done")()
        }
        else{println@Console("")()}
    };
    println@Console("---================---")()
}

init{
    getCurrentTimeMillis@Time()(wi.id);
    wi_stack[#wi_stack]<<wi
}

define create_and_offer_wi{
    undef(wi);
    getCurrentTimeMillis@Time()(wi.id);
    random@Math()(pd);
    pd=int(pd*2);
    if(pd){
        random@Math()(wi.depends);
        wi.depends=int(wi.depends*#wi_stack)
    };
    wi_stack[#wi_stack]<<wi;
    print@Console("Created wi "+wi.id)();
    if(pre_do){println@Console(", depending on: "+wi.depends)()}
    else{println@Console("")()}
}

main{
    {while(true){
        synchronized(wi_lock){
            create_and_offer_wi;
            print_wi_stack
        };
        sleep@Time(1000)()
    }}|{
    while(true){
```

```
50          //DO
51          synchronized(wi_lock){
52              rid1.wi->wi_stack[do_i];
53              rid1.found=false;
54              if(#wi_stack>0){
55                  for(do_i=0,do_i<#wi_stack && !rid1.found,do_i++){
56                      if(!rid1.wi.done && !rid1.wi.taken){
57                          if(!is_defined(rid1.wi.depends) ||
58                              wi_stack[rid1.wi.depends].done){
59                                  rid1.found=true;
60                                  rid1.wi.taken=true;
61                                  do_i--
62              }}}}};
63              sleep@Time(1500)();
64              synchronized(wi_lock){
65                  if(rid1.found){
66                      rid1.wi.done=true;
67                      println@Console("RID1 completed wi: "+
68                          rid1.wi.id)();
69                      undef(rid1.wi)
70                  }
71              }
72          }|
73      while(true){
74          //PRE-DO
75          synchronized(wi_lock){
76              rid1.pre_do.wi->wi_stack[pre_do_i];
77              rid1.pre_do.found=false;
78              if(#wi_stack>0){
79                  for(pre_do_i=0,pre_do_i<#wi_stack &&
80                      !rid1.pre_do.found ,pre_do_i++){
81                      if(!rid1.pre_do.wi.done &&
82                          !rid1.pre_do.wi.taken){
83                          if(!is_defined(rid1.pre_do.wi.depends)){
84                                  rid1.pre_do.found=true;
85                                  rid1.pre_do.wi.taken=true;
86                                  pre_do_i--
87              }}}}};
88              sleep@Time(1000)();
89              synchronized(wi_lock){
90                  if(rid1.pre_do.found){
91                  rid1.pre_do.wi.done=true;
92                  println@Console("RID1 pre-done wi: "+
93                  rid1.pre_do.wi.id)()
94                  }
95              }
96          }
97      }
98  }
```

3.4.8 Auto-Start Patterns

Auto-start patterns relate to situations where execution of work items is triggered by specific events in the lifecycle of the work item or the related process definition. Such events may include the creation or allocation of the work item, completion of another instance of the same work item or a work item that immediately precedes the one in question.

The state transitions associated with these patterns are illustrated by bold arcs in Figure 3.49.

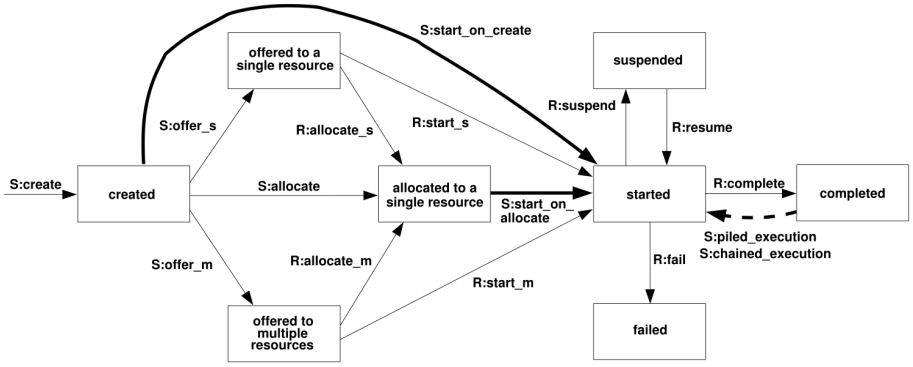


Figure 3.49: Auto-start Patterns

3.4.8.1 Commencement on Creation

The ability for a resource to commence execution on a work item as soon as it is created.

Motivation

The ability to commence execution on a work item as soon as it is created offers a means of expediting the overall throughput of a case, as it removes the delays associated with allocating the work item to a suitable resource and also the time that the work item remains in the resource's work queue, prior to it being started.

JOLIE **Implementation**

Where a task is specified as being subject to *Commencement on Creation*, when the task is initiated in a process instance, the associated work item is created, allo-

cated and commenced simultaneously. This pattern is illustrated by the transition *S:start_on_create* in Figure 3.49.

The JOLIE implementation of this pattern is the same discussed for the *Direct Distribution* pattern [3.4.4.1], in which the creation on the work item is followed directly by the execution of the resource which the work item is allocated to.

3.4.8.2 Commencement on Allocation

The ability to commence execution on a work item as soon as it is allocated to a resource.

Motivation

Although combined creation, allocation and commencement of work items promotes more efficient process throughput, it effectively requires "hard-coding" of resource identities in order to manage work item allocation at creation time.

Commencing work items at the point of allocation does not require resource identity to be predetermined and offers a means of expediting throughput without necessitating changes to the underlying process model.

JOLIE **Implementation**

Where a task is specified as being subject to *Commencement on Allocation*, the act of allocating an associated work item in a process instance also results in it being commenced. In effect, it is put into the work list of the resource to which it is allocated with a started status rather than an allocated status. This pattern is illustrated by the transition *S:start_on_create* in Figure 3.49.

An offering achieves full support if it satisfies the description for the pattern.

The JOLIE implementation of this pattern uses the same structure discussed for the implementation of the *Recursion Pattern* [3.2.8.3]. With this approach, every time a new allocation is done, by the work item creator process, a new resource branch is fired parallelly, thus guaranteeing the achievement of the behavior described for the *Commencement on Allocation* pattern.

JOLIE **code example**

Listing 83: Commencement on Allocation code example

```
include "console.iol"
include "time.iol"
include "math.iol"

outputPort COA{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: rid,create
}

inputPort COA{
```

```
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: rid,create
}

execution{concurrent}

define print_wi_stack{
    println@Console("---=== WI STACK ===---")();
    for(pi=0,pi<#wi_stack,pi++){
        print@Console("id: "+wi_stack[pi].id)();
        if(is_defined(wi_stack[pi].depends)){
            print@Console(", depends from wi: "+
                wi_stack[pi].depends)()
        };
        if(wi_stack[pi].done){
            println@Console(", done")()
        }
        else{println@Console("")()}
    };
    println@Console("---================---")()
}

init{
    wi_stack->global.wi_stack;
    create@COA()
}

define create_and_offer_wi{
    undef(wi);
    getCurrentTimeMillis@Time()(wi.id);
    wi_stack[#wi_stack]<<wi;
    println@Console("Created wi "+wi.id)()
}

main{
    [rid(wid)]{
        sleep@Time(1000)();
        wi_stack[wid].done=true;
        println@Console("RID1 completed wi: "+
                    wi_stack[wid].id)()
    }
    [create()]{
    {while(true){
        random@Math()(new_wi);
        new_wi=int(new_wi*10);
        for(j=0,j<new_wi,j++){
            create_and_offer_wi;
            sleep@Time(500)()
        };
        print_wi_stack;
        sleep@Time(1000)()
    }}|{
```

```
65          while(true){
66              if(#wi_stack>0){
67                  for(i=0,i<#wi_stack,i++){
68                      if(!wi_stack[i].done && !wi_stack[i].started){
69                          wi_stack[i].started=true;
70                          rid@COA(i)
71      }}}}}}
72  }
```

3.4.8.3 Piled Execution

The ability to initiate the next instance of a task (perhaps in a different case) once the previous one has completed, with all associated work items being allocated to the same resource. The transition to *Piled Execution* mode is at the instigation of an individual resource. Only one resource can be in *Piled Execution* mode for a given task at any time.

Motivation

Piled Execution provides a means of optimizing task execution by pipelining instances of the same task and allocating them to the same resource.

JOLIE **Implementation**

Piled Execution involves a resource undertaking work items corresponding to the same task sequentially. These work items may be in different cases. Once a work item is completed, if another work item corresponding to the same task is present in the work queue, it is immediately started.

In effect, the resource attempts to work on piles of the same types of work items. The aim with this approach to work distribution is to allocate similar work items to the same resource, which aims to undertake them one after the other, thus gaining from the benefit of exposure to the same task.

This pattern is illustrated by the transition *R:piled_execution* in Figure 3.49. It is important to note that this transition is represented by a dashed line because it jumps from one work item to another, i.e., it links the life-cycles of two different work items in distinct cases.

To implement this pattern requires the work items to be allocated to the same resource and the ability for the resource to undertake related work items on a sequential basis, immediately commencing the next one when the previous one is complete.

This is a relatively sophisticated requirement and none of the other offerings examined support it. It is included in this taxonomy as it constitutes a logical extension of the concepts that underpin the *Commencement on Creation* pattern [3.4.8.1] enabling instances of the same task across multiple cases to be allocated to a single resource.

An offering achieves full support if it satisfies the description for the pattern.

To implement this kind of pattern, keeping it's realization as strict as possible to the definition given above, the JOLIE code example provided as it follows is divided into two parts:

- client(s), which represents the multiple case instances that create new work items and allocate them to the resource;
- server, which includes both the piled allocation/execution of tasks of the same kind (in this example based on the process architecture) and the (two) resources. It exposes an allocation operation that can be invoked independently by each client. Parallelly the two resources can pick each work item in a sequential basis and run its task until completion.

JOLIE **code example**

Listing 84: Piled Execution (server) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

inputPort PE{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: allocate,runRID1,runRID2
}

outputPort PE{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: runRID1,runRID2
}

execution{concurrent}

define print_wi_stack{
    println@Console("---=== WI STACK ===---")();
    for(pi=0,pi<#wi_stack,pi++){
        print@Console("id: "+wi_stack[pi].id)();
        if(wi_stack[pi].done){
            println@Console(", done")()
        }
        else{println@Console("")()}
    };
    println@Console("---================---")()
}

init{
    wi_stack->global.wi_stack;
```

```
    runRID1@PE()|runRID2@PE()
}

main{
    [runRID1()]{
        while(true){
            synchronized(rid1_lock){
            for(i=0,i<#wi_stack.rid1 && #wi_stack.rid1>0,i++){
                if(!wi_stack.rid1[i].done){
                    sleep@Time(wi_stack.rid1[i].comp_time)();
                    wi_stack.rid1[i].done=true;
                    println@Console("RID1 completed wi: "+
                        wi_stack.rid1[i].id+", arch:"+wi_stack.rid1[i].arch)()
    }}}}}
    [runRID2()]{
        while(true){
            synchronized(rid2_lock){
            for(j=0,j<#wi_stack.rid2 && #wi_stack.rid2>0,j++){
                if(!wi_stack.rid2[j].done){
                    sleep@Time(wi_stack.rid2[j].comp_time)();
                    wi_stack.rid2[j].done=true;
                    println@Console("RID1 completed wi: "+
                        wi_stack.rid2[j].id+", arch:"+wi_stack.rid2[i].arch)()
    }}}}}
    [allocate(wi)]{
        if(wi.arch=="x86"){
            synchronized(rid1_lock){
                wi_stack.rid1[#wi_stack.rid1]<<wi
            }
        }
        else{
            synchronized(rid2_lock){
                wi_stack.rid2[#wi_stack.rid2]<<wi
            }
        }
    }
}
```

Listing 85: Piled Execution (client) code example

```
include "console.iol"
include "time.iol"
include "math.iol"

outputPort PE{
    Location: "socket://localhost:8000"
    Protocol: sodep
    OneWay: allocate
}

init{
```

```
12      random@Math()(arch);
13      arch=int(arch*2);
14      if(arch){arch="x86"}else{arch="x64"};
15      wi.arch=arch
16  }
17
18  define create_and_allocate_wi{
19      getCurrentTimeMillis@Time()(wi.id);
20      random@Math()(wi.comp_time);
21      wi.comp_time=int(wi.comp_time*1000);
22      allocate@PE(wi);
23      println@Console("Created wi "+wi.id+", arch:"+wi.arch+
24          ", comp_time:"+wi.comp_time)()
25  }
26
27  main{
28      while(true){
29          create_and_allocate_wi;
30          sleep@Time(500)()
31      }
32  }
```

3.4.8.4 Chained Execution

The ability to automatically start the next work item in a case once the previous one has completed. The transition to *Chained Execution* mode is at the instigation of the resource.

Motivation

The rationale for this pattern is that case throughput is expedited when a resource has sequential work items allocated within a case and when a work item is completed, its successor is immediately initiated. This has the effect of keeping the resource constantly progressing a given case.

JOLIE **Implementation**

Chained Execution involves a resource undertaking work items in the same case in "chained mode" such that the completion of one work item immediately triggers its successor which is immediately placed in the resource's work list with a started status. This pattern is illustrated by the transition *R:chained_execution* in Figure 3.49.

It is important to note that this transition is represented by a dashed line because it jumps from one work item to another, i.e., it links the life-cycles of two different work items.

Chained Execution offers a means of achieving rapid throughput for a given case however, in order to ensure that this does not result in an arbitrary delay of other cases, it is important that cases are distributed across the widest possible range of resources and that the distribution only occurs when a resource is ready to undertake a new case.

An offering achieves full support if it satisfies the description for the pattern.

The JOLIE code example for this pattern is obtained by linking a case value to each work item which, once found, instigates the resource to complete, sequentially, each work item that's present in the work queue within that case.

After all work items for that case are exhausted, the case of the next (not done) work item is taken and the *Chained Execution* is repeated.

JOLIE **code example**

Listing 86: Chained Execution code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define print_wi_stack{
    println@Console("---=== WI STACK ===---")();
    for(pi=0,pi<#wi_stack,pi++){
        print@Console("id: "+wi_stack[pi].id)();
        print@Console(", of case: "+
                wi_stack[pi].c_case)();
        if(wi_stack[pi].done){
            println@Console(", done")()
        }
        else{println@Console("")()}
    };
    println@Console("---================---")()
}

define create_and_offer_wi{
    undef(wi);
    getCurrentTimeMillis@Time()(wi.id);
    random@Math()(wi.c_case);
    wi.c_case=int(wi.c_case*10);
    wi_stack[#wi_stack]<<wi
}

define chained_execution{
    println@Console("Chained execution on case: "+ce.c_case)();
    for(ci=0,ci<#wi_stack,ci++){
        if(!wi_stack[ci].done &&
            wi_stack[ci].c_case==ce.c_case){
            sleep@Time(500)();
            wi_stack[ci].done=true
        }
    }
}

main{
    {while(true){
        synchronized(wi_lock){
            for(cr=0,cr<20,cr++){
                create_and_offer_wi
            }
        };
        sleep@Time(5000)()
    }}|{
    while(true){
        //CHAINED EXECUTION
        if(#wi_stack>0){
```

```
50              for(i=0,i<#wi_stack,i++){
51                  if(!wi_stack[i].done){
52                  synchronized(wi_lock){
53                      ce.c_case=wi_stack[i].c_case;
54                      chained_execution;
55                      print_wi_stack
56                  }
57      }}}}}
58  }
```

3.4.9 Visibility Patterns

Visibility Patterns classify the various scopes in which work item availability and commitment are able to be viewed by resources.

3.4.9.1 Configurable Unallocated Work Item Visibility

The ability to configure the visibility of unallocated work items by process participants.

Motivation

The pattern denotes the ability to limit the visibility of unallocated work items, either to potential resources to which they may subsequently be offered or allocated to, or to completely shield knowledge of created but not yet allocated work items from all resources.

JOLIE **Implementation**

The ability to view unallocated work items is usually implemented as a configurable option on a per-user basis.

Of most interest is the ability to view work items in an offered state.

The JOLIE implementation of this pattern is archived by the language's tree-like data structures which enables to create easily taxonomies and scoped views, according to the visibility each resource is allowed to have over the work items in the system.

In the example of structure provided as it follows - created by means of the `with` statement of the language, which implements a shortcut for repetitive variable paths -, a resource whose scope is set to the whole structure has the complete visibility over all of the work items present in the stack.

The navigation among all of the elements (from the root to a leaf and vice-versa) of the structure can be both static and dynamic, by means of JOLIE's dynamic look-up feature.

Other resource that shall have a restricted view over the whole stack of work items are able only to navigate within a branch of the whole structure, e.g. the resource RID1 shall be able to browse the `wi_stack.rid1` branch only, likewise the resource RID2 will be allowed to do the same only over its own branch `wi_stack.rid2`.

Each item's view remains highly configurable since structure alias (->) and deep-copy (<<) statements can be used to temporary grant access to a scoped resource to view and interact with a whole sub-structure or a single work item.

JOLIE **code example**

Listing 87: Configurable Unallocated Work Item Visibility code example

```
include "console.iol"

main{
with (wi_stack){
    .created[0]="8205930411";
    .created[1]="5450147330";
    .created[2]="4463849723";

    .rid1[0]="8265330677";
    .rid1[1]="5473457670";

    .rid2[3]="3457654532";
    .rid2[4]="3138763121";
}}
```

3.4.9.2 Configurable Allocated Work Item Visibility

The ability to configure the visibility of allocated work items by process participants.

Motivation

The pattern indicates the ability to limit the visibility of allocated and started work items.

JOLIE **Implementation**

The ability to view allocated work items is usually implemented as a configurable option on a per-user basis. It provides resources with the ability to view work items in an allocated or started state.

An offering achieves full support if it provides a construct that satisfies the description for the pattern.

The JOLIE implementation of this pattern is equal to the one provided for the previous *Configurable Unallocated Work Item Visibility* pattern [3.4.9.1], thus is made reference to the one provided for the latter.

3.4.10 Multiple Resource Patterns

In situations where people are not restricted by information technology, there is often a many-to-many correspondence between the resources and work items in a given allocation or execution. Therefore, it may be desirable to support this using process technology.

Simultaneous Execution is a one-to-many correspondence between the resources and work items in a given allocation or execution. The opposite approach, the many-to-one correspondence, i.e. multiple resources working on the same work item, is more difficult to achieve and poses some concerns about work division and work item state preservation. This is a typical situation of activities in which people tend to work in teams and collaborate to jointly executed work items.

3.4.10.1 Simultaneous Execution

The ability for a resource to execute more than one work item simultaneously.

Motivation

In many situations, a resource does not undertake work items allocated to it on a sequential basis, but rather it commences work on a series of work items and multitasks between them.

JOLIE **Implementation**

The *Simultaneous Execution* pattern recognizes more flexible approaches to work item management, where the decision as to which combination of work items will be executed and the sequence in which they will be interleaved is at the discretion of the resource rather than the system.

An offering achieves full support if it satisfies the description for the pattern. It achieves a partial support rating if there are any limitations on the range of work items that can be executed simultaneously.

An offering achieves full support if it satisfies the description for the pattern.

The JOLIE code example for this pattern is provided by the implementation of a resource which have to process several steps of a single work item to mark it as completed.

Instead of dedicating all of its computation time to the completion of one work item only (single burst), the resource tries to work on as many work items as possible, completing one of the "not-done" steps of each work item present in its queue (multitask).

JOLIE **code example**

Listing 88: Simultaneous Execution code example

```
include "console.iol"
include "time.iol"
include "math.iol"

define print_wi_stack{
    println@Console("---=== WI STACK ===---")();
    for(pi=0,pi<#wi_stack,pi++){
        print@Console("id: "+wi_stack[pi].id)();
        if(wi_stack[pi].done){
            println@Console(", done")()
        }else{
        println@Console()();
        for(ri=1,ri<4,ri++){
            print@Console("\tstep "+ri+", comp_time: "+
                wi_stack[pi].step[ri].comp_time
                +"ms")();
                if(wi_stack[pi].step[ri].done){
                    println@Console(", done")()
                }else{println@Console()()}
    }}};
    println@Console("---================---")()
}


define create_and_offer_wi{
    getCurrentTimeMillis@Time()(wi.id);
    for(ci=0,ci<4,ci++){
        random@Math()(comp_time);
        wi.step[ci].comp_time=int(comp_time*1000)
    };
    wi_stack[#wi_stack]<<wi;
    println@Console("Created wi "+wi.id)()
}

main{
    {while(true){
        synchronized(wi_lock){
            create_and_offer_wi;
            print_wi_stack
        };
        sleep@Time(1000)()
    }}|{
```

```
43      while(true){
44          for(i=0,i<#wi_stack,i++){
45              if(!wi_stack[i].done){
46                  println@Console("Working on "+wi_stack[i].id)();
47                  step_done=false;
48                  for(j=0,j<#wi_stack[i].step && !step_done,j++){
49                      if(!wi_stack[i].step[j].done){
50                          sleep@Time(wi_stack[i].step[j].comp_time)();
51                          synchronized(wi_lock){
52                          wi_stack[i].step[j].done=true;
53                          if(j==#wi_stack[i].step-1){
54                          wi_stack[i].done=true
55                          }};
56                          step_done=true
57      }}}}}}
58  }
```

3.4.10.2 Additional Resources

The ability for a given resource to request additional resources to assist in the execution of a work item that it is currently undertaking.

Motivation

In more complex scenarios, a given work item may require the services of multiple resources in order for it to be completed (e.g. a machine operator, machine and fuel).

These resources may be durable in nature and capable of continual reuse or they may be consumable. By providing the ability to model scenarios such as these, a more accurate depiction of the way in which work is actually undertaken in a production environment is made possible.

JOLIE **Implementation**

This pattern recognizes more complex work distribution and resource management scenarios where simply unitary resource allocation is not sufficient to deal with the constraints that tasks may experience during execution.

An offering achieves full support if it provides a construct that satisfies the description for the pattern. It achieves a partial support rating if there are limitations on the situations in which multiple resources can be modeled or utilized.

The JOLIE realization for this pattern is provided by the presence of two resources: RID1 which is the "main" resource, that has the capability to call for *Additional Resources* to join it's work queue and help it executing all of its tasks. Contrariwise, RID2 is a callable (additional) resource, which has the capability to collaborate in completing RID1 work items' steps.

The implementation of this pattern is provided by means of a "shared" state variable linked to each work item, in this way resources allowed or dedicated to execute shared work items can access and work on them.

JOLIE **code example**

Listing 89: Additional Resources code example

```
1 include "console.iol"
2 include "time.iol"
3 include "math.iol"
4
```

```
define print_wi_stack{
    println@Console("---=== WI STACK ===---")();
    for(pi=0,pi<#wi_stack.rid1,pi++){
        print@Console("id: "+wi_stack.rid1[pi].id)();
        if(wi_stack.rid1[pi].done){
            println@Console(", done")()
        }else{
        println@Console()();
        for(ri=1,ri<4,ri++){
            print@Console("\tstep "+ri+", comp_time: "+
                wi_stack.rid1[pi].step[ri].comp_time
                +"ms")();
                if(wi_stack.rid1[pi].step[ri].done){
                    println@Console(", done by "+
                    wi_stack.rid1[pi].step[ri].by)()
                }else{println@Console()()}
    }}};
    println@Console("---===============---")()
}

define create_and_offer_wi{
    getCurrentTimeMillis@Time()(wi.id);
    for(ci=0,ci<4,ci++){
        random@Math()(comp_time);
        wi.step[ci].comp_time=int(comp_time*1000)
    };
    wi_stack.rid1[#wi_stack.rid1]<<wi;
    println@Console("Created wi "+wi.id)()
}

define add_resource{
    done_wi=0;undone_wi=0;
    for(adi=0,adi<#wi_stack.rid1,adi++){
        if(wi_stack.rid1[adi].done){
            done_wi++
        }else{undone_wi++}
    };
    if(undone_wi>((3/4)*done_wi)+1){
        for(adi=0,adi<#wi_stack.rid1,adi++){
        if(!wi_stack.rid1[adi].shared
            && !wi_stack.rid1[adi].done){
            wi_stack.rid1[adi].shared=true;
            println@Console("Added wi:"+
                wi_stack.rid1[adi].id+
                " to RID2 queue")()
}}}}

main{
    {while(true){
        synchronized(wi_lock){
            create_and_offer_wi;
            print_wi_stack
```

```
57          };
58          sleep@Time(1500)()
59      }}|{
60      while(true){
61          for(i=0,i<#wi_stack.rid1,i++){
62              if(!wi_stack.rid1[i].done){
63                  synchronized(wi_lock){add_resource};
64                  println@Console("RID1 Working on "+
65                      wi_stack.rid1[i].id)();
66                  step_done=false;
67                  for(j=0,j<#wi_stack.rid1[i].step && !step_done,j++){
68                      if(!wi_stack.rid1[i].step[j].done){
69                          sleep@Time(wi_stack.rid1[i].step[j].comp_time)();
70                          synchronized(wi_lock){
71                          wi_stack.rid1[i].step[j].done=true;
72                          wi_stack.rid1[i].step[j].by="RID1";
73                          if(j==#wi_stack.rid1[i].step-1){
74                          wi_stack.rid1[i].done=true
75                          }};
76                          step_done=true
77      }}}}}}|{
78      while(true){
79          for(x=0,x<#wi_stack.rid1,x++){
80              if(!wi_stack.rid1[x].done
81                   && wi_stack.rid1[x].shared){
82                  println@Console("RID2 Working on "+
83                      wi_stack.rid1[x].id)();
84                  step_done=false;
85                  for(y=0,y<#wi_stack.rid1[x].step && !step_done,y++){
86                      if(!wi_stack.rid1[x].step[y].done){
87                          sleep@Time(wi_stack.rid1[x].step[y].comp_time)();
88                          synchronized(wi_lock){
89                          wi_stack.rid1[x].step[y].done=true;
90                          wi_stack.rid1[x].step[y].by="RID2";
91                          if(y==#wi_stack.rid1[x].step-1){
92                          wi_stack.rid1[x].done=true
93                          }};
94                          step_done=true
95      }}}}}}
96  }
```

3.5 Summary Table of JOLIE Resource Patterns Support

Resource Pattern	JOLIE **Support**
Direct Distribution [3.4.4.1]	+
Role-Based Distribution [3.4.4.2]	+
Deferred Distribution [3.4.4.3]	+
Authorization [3.4.4.4]	+/-
Separation of Duties [3.4.4.5]	+/-
Case Handling [3.4.4.6]	+
Retain Familiar [3.4.4.7]	+
Capability-Based Distribution [3.4.4.8]	+
History-Based Distribution [3.4.4.9]	+
Organizational Distribution [3.4.4.10]	+
Automatic Execution [3.4.4.11]	+
Distribution by Offer - Single Resource [3.4.5.1]	+
Distribution by Offer - Multiple Resources [3.4.5.2]	+
Distribution by Allocation - Single Resource [3.4.5.3]	+
Random Allocation [3.4.5.4]	+
Round Robin Allocation [3.4.5.5]	+
Shortest Queue [3.4.5.6]	+
Early Distribution [3.4.5.7]	+
Distribution on Enablement [3.4.5.8]	+
Late Distribution [3.4.5.9]	+

Table 3.3: The table provided lists all of Resource patterns analyzed previously. Following the same convention adopted by WPI if a pattern can be realized in JOLIE directly it is rated +, if is not directly supported, but has been realized through a workaround is rated +/-, finally if no implementation is supported is rated -.

Resource Pattern	JOLIE **Support**
Resource-Initiated Allocation [3.4.6.1]	+
Resource-Initiated Execution - Allocated Work Item [3.4.6.2]	+
Resource-Initiated Execution - Offered Work Item [3.4.6.2]	+
System-Determined Work Queue Content [3.4.6.4]	+
Resource-Determined Work Queue Content [3.4.6.5]	+
Selection Autonomy [3.4.6.6]	+
Delegation [3.4.7.1]	+
Escalation [3.4.7.2]	+
Deallocation [3.4.7.3]	+
Stateful Reallocation [3.4.7.4]	+
Stateless Reallocation [3.4.7.5]	+
Suspension-Resumption [3.4.7.6]	+
Skip [3.4.7.7]	+
Redo [3.4.7.8]	+
Pre-Do [3.4.7.9]	+
Commencement on Creation [3.4.8.1]	+
Commencement on Allocation [3.4.8.2]	+/-
Piled Execution [3.4.8.3]	+
Chained Execution [3.4.8.4]	+
Configurable Unallocated Work Item Visibility [3.4.9.1]	+
Configurable Allocated Work Item Visibility [3.4.9.2]	+
Simultaneous Execution [3.4.10.1]	+
Additional Resources [3.4.10.2]	+

Table 3.4: The table provided lists all of Resource patterns analyzed previously. Following the same convention adopted by WPI if a pattern can be realized in JOLIE directly it is rated +, if is not directly supported, but has been realized through a workaround is rated +/-, finally if no implementation is supported is rated -.

Chapter 4

Conclusions

4.1 On Workflow Patterns & JOLIE

The Third Chapter proves the high customizability of the solutions composable with the SOC approach and JOLIE.

It's interesting to take into account that, as aforementioned in JOLIE's patterns support summary tables, the WPI did not only enunciate an wide and finely declined plethora of patterns, organized in four areas according to their role in system designing, but evaluated an extensive list of workflow products, either commercial, open-source or (proposed) standards, against the feasibility of its own workflow patterns. Finally each evaluation has been used to compile a general list for comparison purposes.

Based on this list, whose rating is the same applied for the evaluation tables of JOLIE, the bobble chart in Figure 4.1 has been created to summarize and provide a means to realized at glance, the degree of adaptability of JOLIE compared to the other language.

The bobble chart reported has been obtained by taking all of the evaluation of languages provided by the WPI, whose summary tables were compiled both for Control-Flow and Resource patterns.

Then, the "+, +/-, -" label rating has been converted into a countable "1, 1/2, 0" rating, where concordantly, 1 is assigned for each "+" (full support) rating, while 1/2 replaces the "+/-" (half support) rating and 0 replaces "-" (no support).

Finally, for each language has been compiled a summary table, reporting an aggregate number linked to the support degree of that language against a specific kind of patterns. This evaluations is made by summing the rating values (1 and 1/2) of a specific kind of patterns, and diving it by the total number of those patterns, ultimately this value is put into percentage.

As aforementioned, the bobble chart is the result of the percentage values of any language where the center of each language's bobble is set in the (x,y) coordinates that correspond to the percentage of support of the specific kind of patterns by the language - x for Control-flow patterns and y for Resource patterns. The area of each bobble represents the total number of patterns supported by each language.

Thanks to this chart, it's a lot easier to understand the general degree of support given by a specific language and it becomes even easier to understand if a language is more suitable for resource management rather than for control-flow purposes.

Clearly JOLIE appears as the all-winning competitor of this challenge, but it's better not to draw hasty conclusions and to think about the work conducted on the language.

The main difference between JOLIE and any other language taken into account by the WPI (and thus, the chart), is that JOLIE is a service-oriented imperative language whose main area is not BPM but SOC.

The difference between these two areas is clearly defined in section 1.1, and, although an high degree of overlap stands between the two, it would be better not to merge them.

On the contrary, BPM languages are actually used to implement SOC solutions, but, as demonstrated, they do not reach the same level of customizability and adaptation of a service oriented language. Furthermore BPM languages used for SOC purposes suffer form behavior realization uncertainty of the functionality defined at design-time. This behavior is caused by the declarative approach of these languages, which requires engines that interpret the solutions proposed by means of not-always-standard heuristics. Such interpretation is sometime unclear and different engines made by different organization may bring to different behaviors, although based on the same language solution.

The conclusions that can be drawn from the work done on workflow patterns and JOLIE, confirm that, to maximize the results from SOC (and SOA) integration, is

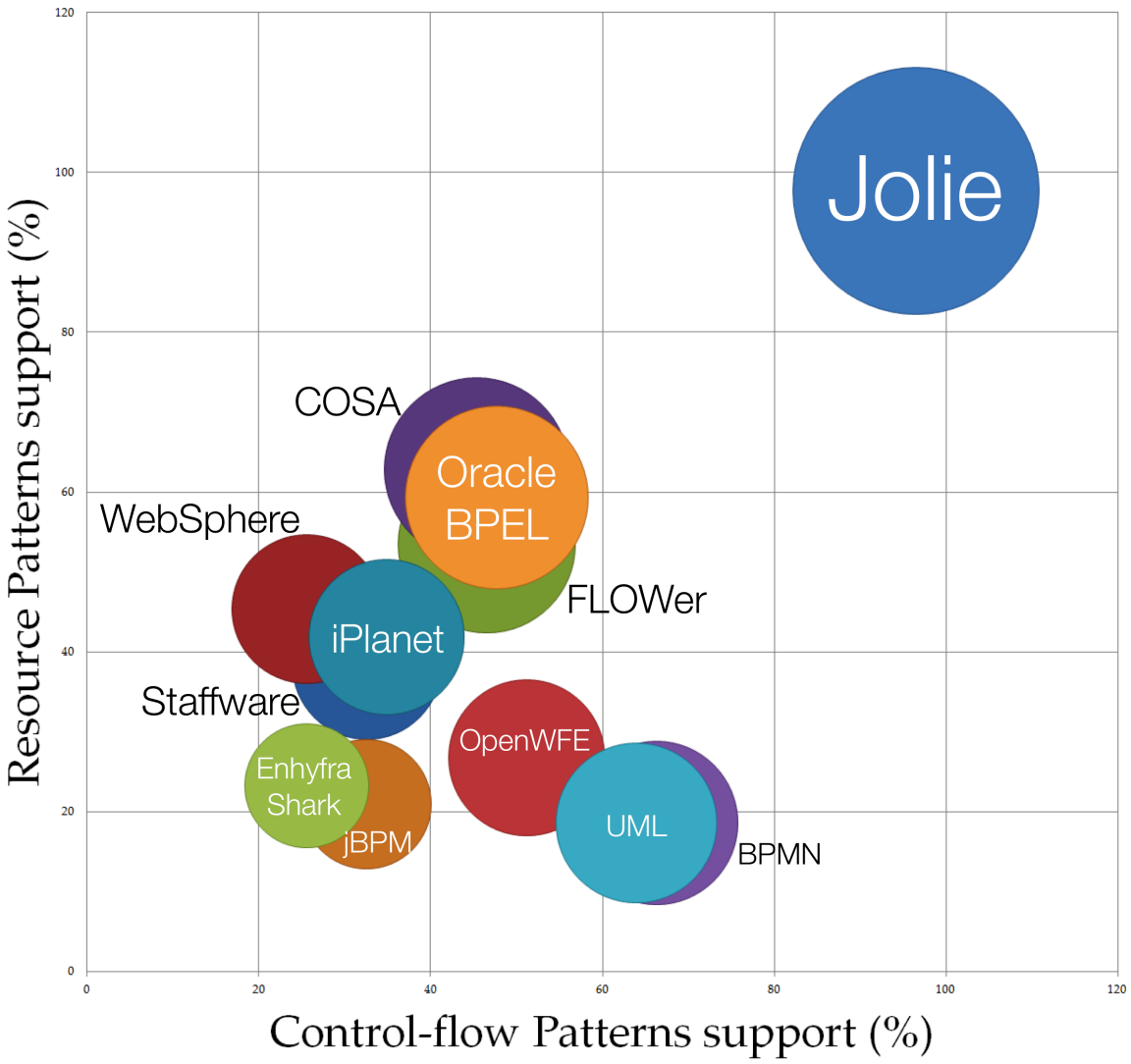


Figure 4.1: Languages Patterns Support

necessary to develop and use specific languages designed on its principles.

This is not a criticism about BPM languages, but its intended as recall of the principle of appropriateness: if satisfactory results are to be achieved, it's fundamental to understand what is the context of the sought solution, and therefore, adopt an approach strictly linked to the rules and principles of that specific field of work.

BPM languages have proved to be extremely useful in the context of business abstract modeling, but, as a matter of facts, they all fall short when running their models as composition of services, due to both uncertainty of results and language constraints.

4.1.1 Future works

The next step of this work shall be the completion of JOLIE support evaluation with relation to the remaining patterns (data and exception handling). Finally the whole work should to be validated from the WPI both to confirm the results obtained and to highlight the possible lack of functionality by an outer organization from the one leading the development project.

Once all of the implementations have been validated, a more business-oriented realization can be assembled, to take advantage of the wide range of language's advanced functionality, not used in this work for simpleness purposes. Done that, the patterns can be employed in the constitution of an extremely useful knowledge base of solutions, which can be applied by developers to structure their own systems, according to the requirements of their models.

4.2 On JOLIE language

JOLIE is a young but powerful and highly adaptive language, which can be used to easily achieve complex results.

As proved in this work, the language, even at its basics, provides both a strong theoretical structure (via SOCK) and a wide range of constructs that overcome the performances of the most part of languages used for SOC purposes.

However a lot of work on the development and evolution of the language has to be done but, it's extremely important to keep in mind that, besides of a solid theoretical basis, any successful project need the support of a strong community.

Thus the work on the language must be undoubtedly carried on together with the constitution of a strong community of developers and users of the language, since, as seen in many other open-source projects, community support and extensive viral spread of use of a language are the springboard for the success of the project itself and, finally, its adoption in the world of industry.

4.3 On Service Oriented Computing and Business Process Automation

The evolution of the SOC is still in progress and the development of JOLIE and other technologies which support this new paradigm is only a small part of a larger revolution that will lead to the next evolutionary step of BPA.

In this sense, the efforts of W3C and other leading open-source and industry associations are producing more and more "intelligent" tools for the computational logic, semantics and business management.

In the near future business processes will seamlessly couple to achieve the goals proposed by their companies, in an autonomous and "conscious" manner, boosting storage, management and fusion of diverse and heterogeneous knowledge in order to let emerge non trivial and strategic information.

The human beings in charge, those who need to take important strategic decisions, can make use of these tools to focus their cognitive skills on real business problems and not, as today, on issue connected to processes realization and limitations derived from data overload and the effort required to manage it.

Systems endowed with semantic and computational logic can "understand" each other, compose themselves and respond, within a few seconds, to questions that, in contrast, have an high impact in the organization's budget (such as quality control, monitoring of business functions, etc.).

As aforementioned this new step in BPA will "augment" the skills of business management and even of each "service" that composes a firm. Helping man in

information management, BPA is the key to manage a fast-changing world whose complexity requires a more integrated synergy between human decision-making systems and information systems.

To boldly go, where no business process has gone before.

Acknowledgments

"It's the end of the world as we know it and I feel fine"

R.E.M.

I dare to cite the R.E.M. and the last line of the refrain which names their homonymous song, why? Because preparing for the future always means looking back, wrapping up your things, taking a deep breath and jumping almost blindfolded in a totally different and unknown world. And I think this case applies perfectly.

But that's ok, it's the life, and I think a life is worth living if, left behind your uncertainties, you can "*feel fine*" even if it's the end of the world, as you've known it.

That's all about change, evolution, discovery and passion for life itself.

Back to the main subject, these are the acknowledgments, maybe the most read part of any thesis, probably second only to the cover and the dedication. And the scientific topic? Leave it to those freaky nerds like... me?! Ouch, it hurt. Let's go on.

My acknowledgments starts with the smallest, unique and most important of all the communities I belong to: my Family.

Mom, Dad, I do not really know how to thank you for all the sacrifices you've done to raise me. The older I get, the more I understand that parenting is the hardest job in the world. But you tackled it, like Olympic champions, and renounced to the most part of your passions to make me the most loved of yours. Thank you for favoring my aspirations, feeding my appetite for knowledge and for letting me draw my own path, constantly suggesting the best for me.

Thank you for all of this and even more, I hope to make you proud of me, every day of my life.

To my little, sweet, lovely Danny.

You complete me, you link me to my deepest humanity and you make me appreciate the life for all the little, silly and irrational things it's made by. Thank you, thank you, thank you. My beloved fellow traveler in this journey called existence, I hope to pursue my adventure with you, for still a long, long time.

Next I want to thank the professors of the University of Bologna and in particular those who deeply believed and gave birth to a no-ordinary course like Scienze di Internet. I thank you for giving me the chance to work, learn and above all "grow up" in a multidisciplinary context such the one of Scienze di Internet.

I want to thank all the people linked to JOLIE, for having created such an interesting, innovative and "cool" project and for having accepted me in it. In particular I want to thank my supervisor, Prof. Maurizio Gabbrielli, Fabrizio Montesi and Claudio Guidi for all the time dedicated to me and the interesting discussions about the project and our future works.

Finally I want to thank all friends of mine for all of the adventures, the passionate confabulations and world-conquering plans we assembled in all these years. I swear you guys, without you, life would be really, really boring.

Ringraziamenti

"It's the end of the world as we know it and I feel fine"

R.E.M.

Mi sono permesso di citatare i R.E.M e l'ultima riga del ritornello dell'omonima canzone, perché? Perché prepararsi al futuro riguarda sempre il guardarsi indietro, raccogliere le proprie cose, prendere un profondo respiro e fare un salto nel buio in un mondo totalmente differente e sconosciuto. E penso che questa circostanza si applichi perfettamente.

Ma va bene, è la vita, e penso che una vita valga la pena di essere vissuta se, abbandonate le proprie incertezze, ci si "sente bene" anche se è la fine del mondo, per come lo hai conosciuto.

Tutto ciò riguarda il cambiamento, l'evoluzione, la scoperta e la passione per la vita stessa.

Tornando all'argomento principale, questi sono i ringraziamenti, forse la parte più letta di ogni tesi, probabilmente seconda solo dopo la copertina e la dedica. E l'argomento scientifico? Lasciamolo a quei bizzarri nerd come… me?! Questa ha fatto male. Andiamo avanti.

I miei ringraziamenti iniziano con la più piccola, unica e importante delle comunità di cui faccio parte: la mia Famiglia.

Mamma, Papà, non so veramente come ringraziarvi per tutti i sacrifici fatti per crescermi. Più avanzo negli anni e più capisco come fare il genitore sia il lavoro più difficile del mondo. Ma voi l'avete affrontato, da campioni olimpici, ed avete rinunciato alla maggior parte delle vostre passioni per fare di me la principale.

Grazie per incoraggiare le mie aspirazioni, per nutrire la mia voglia di conoscenza e per lasciarmi tracciare il mio cammino, elargendomi sempre i migliori consigli.

Grazie per tutto questo e tanto altro ancora, spero di rendervi fieri di me, ogni giorno della mia vita.

Alla mia piccola, dolce, amorevole Danny.

Tu mi completi, mi colleghi alla mia umanità più profonda e mi fai apprezzare la vita per tutte le piccole, stupide ed irrazionali cose di cui è fatta. Grazie, grazie, grazie. Mia amata compagna di viaggio, spero di proseguire con te quest'avventura per ancora molto, molto tempo.

In seguito voglio ringraziare i professori dell'Università di Bologna e in particolare quelli che hanno creduto profondamente e dato la luce ad un corso inconsueto come Scienze di Internet. Vi ringrazio per avermi dato la possibilità di lavorare, imparare e soprattutto "crescere" con voi in un tale ambito multidisciplinare .

Voglio ringraziare anche tutte le persone collegate a JOLIE, per aver creato un progetto così interessante, innovativo e "figo" e per avermi accolto in esso. In particolare voglio ringraziare il mio relatore, il Prof. Maurizio Gabbrielli, Fabrizio Montesi e Claudio Guidi per tutto il tempo dedicatomi e le interessanti discussioni sul progetto e i nostri lavori futuri.

Infine voglio ringraziare tutti i miei amici per tutte le avventura, le appassionanti confabulazioni e i piani di conquista del mondo che abbiamo concepito in tutti questi anni. Ve lo giuro ragazzi, senza di voi, la vita sarebbe molto ma molto noiosa.

Bibliography

[1] F. Daniel, B. Pernici (2006), *Insights into Web Service Orchestration and Choreography*, International Journal of E-Business Research, 2(1), 58-77.

[2] Y. Vasiliev (2007), *SOA and WS-BPEL Composing Service-Oriented Solutions with PHP and ActiveBPEL*, Packt Publishing.

[3] C. Peltz (2003), *Web Services Orchestration and Choreography*, IEEE Computer Society.

[4] J. Mendling, M. Hafner, *From WS-CDL Choreography to BPEL Process Orchestration*, Vienna University of Economics and Business Administration - WU Wien, Austria.

[5] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro (2006), *Choreography and Orchestration conformance for system design*, Department of Computer Science, University of Bologna, Italy.

[6] A. Barros, M. Dumas, P. Oaks (2005), *A Critical Overview of the Web Services Choreography Description Language (WS-CDL)*, BPTrends.

[7] M. zur Muehlen, J.V. Nickerson, K.D. Swenson (2004), *Developing Web Services Choreography Standards The Case of REST vs. SOAP*, Decision Support Systems 37, Elsevier, North Holland.

[8] C. Peltz (2003), *Web Services Orchestration and Choreography - A look at WSCI and BPEL4WS*, IEEE Computer Society.

[9] C. Abrams, R.W. Schulte (2008), *Service-Oriented Architecture Overview and Guide to SOA Research*, Gartner.

[10] T. Erl (2009), *SOA design patterns*, SOA Systems Inc.

[11] T. Erl (2008), *SOA Principles of Service Design*, SOA Systems, Inc.

[12] M.P. Singh, M.N. Huhns (2005), *SERVICE-ORIENTED COMPUTING Semantics, Processes, Agents*, John Wiley & Sons Ltd.

[13] D. Georgakopoulos and M. P. Papazoglou (2009), *Service-Oriented Computing*, The MIT Press.

[14] C. Guidi, R. Lucchi, G. Zavattaro, N. Busi, R. Gorrieri (2006), *SOCK: a calculus for service oriented computing*, Technical Report UBLCS-2006-20.

[15] K. Ryan, L. KoA (2009), *Computer scientist's introductory guide to business process management (BPM)*, ACM Digital Library.

[16] S. Giallorenzo (2012), *Java Orchestration Language Interpreter Engine - a Tutorial for the Service-Oriented JOLIE language.*

[17] F. Montesi, C. Guidi, G. Zavattaro (2010), *JOLIE: a Service-oriented Programming Language*, Alma Mater Studiorum Università di Bologna.

[18] F. Montesi, C. Guidi, R. Lucchi, G. Zavattaro (2006), *JOLIE: a Java Orchestration Language Interpreter Engine*, Electronic Notes in Theoretical Computer Science.

[19] P. Anedda, M. Gaggero, S. Manca (2008), *A general Service Oriented Approach for managing virtual machines allocation*, SAC'09.

[20] F. Montesi, C. Guidi, G. Zavattaro (2007), *Composing services with JOLIE, Department of Computer Science*, University of Bologna, Italy.

[21] F. Montesi, C. Guidi, I. Lanese and G. Zavattaro, *Dynamic fault handling mechanisms for service-oriented applications*, Department of Computer Science, University of Bologna, Italy.

[22] C. Guidi, F. Montesi (2009), *Reasoning About a Service-oriented Programming Paradigm*, M.H. ter Beek (Ed.): Young Researchers Workshop on Service-Oriented Computing.

[23] W.M.P. van der Aalst, C. Stahl (2011), *Modeling Business Processes – A Petri Net-Oriented Approach*, The MIT Press.

[24] K. Jensen, L. M. Kristensen, L. Wells (2007), *Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems*, Springer-Verlag.

[25] K. Jensen (1997), *A Brief Introduction to Coloured Petri Nets*, Computer Science Department, University of Aarhus.

[26] W. Nauber (2010), *Design and Analysis with Petri Nets*, Technische Universität Dresden Faculty Of Computer Science.

[27] K. Jensen (1998), *An Introduction to the Practical Use of Coloured Petri Nets*, Department of Computer Science, University of Aarhus.

[28] W.M.P. van der Aalst, K.M. van Heem, G.J. Houben (2000), *Modelling and analysing workflow using a Petri-net based approach*, Eindhoven University of Technology, Dept. of Mathematics and Computing Science.

[29] K. Jensen, Lars M. Kristensen (2009),*Coloured Petri Nets Modelling and Validation of Concurrent Systems*, Springer.

[30] A. H. M. ter Hofstede, W.M. P. van der Aalst, M. Adams, N. Russell (2009), *Modern Business Process Automation YAWL and its Support Environment*, Springer.

[31] N.A. Mulyar and W.M.P. van der Aals (2005), *Patterns In Colored Petri Nets*, Department of Technology Management, Eindhoven University of Technology.

[32] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, N. Mulyar (2006), *Workflow Control-Flow Patterns : A Revised View*, BPM Center Report BPM-06-22.

[33] W.M.P van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, A.P. Barros (2003), *Workflow Patterns*, Distributed and Parallel Databases.

[34] B. Kiepuszewski, A.H.M. ter Hofstede, C. Bussler (2000), *On Structured Workflow Modelling*, Proceedings Twelfth International Conference on Advanced Information Systems Engineering (CAiSE'), volume 1789 of Lecture Notes in Computer Science, pages 431-445, Springer Verlag.

[35] B. Kiepuszewski, A.H.M. ter Hofstede, W.M.P. van der Aalst (2003), *Fundamentals of Control Flow in Workflows*, Acta Informatica, 39(3):143-209.

[36] P. Wohed, W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, N. Russell (2006), *Pattern-based Analysis of BPMN - An extensive evaluation of the Control-flow*, the Data and the Resource Perspectives (revised version), BPM Center Report.

[37] P. Wohed, W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede (2003), *Analysis of Web Services Composition Languages: The Case of BPEL4WS*, In I.Y. Song, S.W. Liddle, T.W. Ling, and P. Scheurmann, editors, 22nd International Conference on Conceptual Modeling, volume 2813 of Lecture Notes in Computer Science, pages 200-215. Springer-Verlag.

[38] W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede (2003), *Web Service Composition Languages: Old Wine in New Bottles?*, In G. Chroust and C. Hofer, editors, Proceedings of the 29th EUROMICRO Conference: New Waves in System Architecture, pages 298-305. IEEE Computer Society.

[39] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, A.P. Barros (2000), *Advanced Workflow Patterns*, In O. Etzion and P. Scheuermann, editors, 7th International Conference on Cooperative Information Systems, volume 1901 of Lecture Notes in Computer Science, pages 18-29. Springer-Verlag.

[40] N. Russell, A.H.M. ter Hofstede, D. Edmond (2004), and W.M.P. van der Aalst, *Workflow Resource Patterns*, BETA Working Paper Series, WP 127, Eindhoven University of Technology.

[41] Workflow Patterns Initiative, *Workflow Patterns home page*, `http://www.workflowpatterns.com`.

[42] JOLIE, *JOLIE: Java Orchestration Language Interpreter Engine*, `http://www.workflowpatterns.com`.

[43] W3C Web Services Choreography Description Language, *Web Services Choreography Description Language Version 1.0*, `http://www.w3.org/TR/ws-cdl-10/`.

[44] W3C Web Services Description Language, *Web Services Description Language (WSDL) 1.1*, `http://www.w3.org/TR/wsdl`.

[45] W3C Web Services Architecture, *Web Services Architecture*, `http://www.w3.org/TR/ws-arch/`.

[46] Semantic Web, Semantic Web, `http://semanticweb.org`.