A Tuplespace-Based Execution Model for Decentralized Workflow Enactment

– Applied to BPEL –

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Daniel Martin

aus Lindau a.B.

Hauptberichter: Prof. Dr. rer. nat. Frank Leymann

Mitberichter: A.o. Univ. Prof. Dr. Dipl.-Ing. Eva Kühn

Tag der mündlichen Prüfung: 04. Mai 2010

Institut für Architektur von Anwendungssystemen

der Universität Stuttgart

2010

Martin, Daniel:
A Tuplespace-Based Execution Model for Decentralized Workflow Enactment : Applied to BPEL / Daniel Martin. –
Als Ms. gedr.. – Berlin : dissertation.de – Verlag im Internet GmbH, 2010
Zugl.: Stuttgart, Univ., Diss., 2010
ISBN 978-3-86624-495-5

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

dissertation.de – Verlag im Internet GmbH 2010

Es wird ausschließlich chlorfrei gebleichtes
Papier (TCF) nach DIN-ISO 9706 verwendet.
Printed in Germany.

dissertation.de - Verlag im Internet GmbH

URL: http://www.dissertation.de

Contents

Zusammenfassung

Die Fähigkeit mit Änderungen umzugehen ist eine der größten Herausforderungen denen Unternehmen heutzutage begegnen. Maßnahmen um die Agilität zu steigern, wie beispielsweise Outsourcing und Offshoring von Teilen eines Unternehmens, befinden sich derzeit schon in Anwendung und werden auch in Zukunft eine wichtige Rolle spielen. Eine Änderung an der Organisation eines Unternehmens zieht jedoch auch immer Änderungen der darunterliegenden IT Infrastruktur nach sich, welche folglich auf eine Art und Weise entworfen sein muss, die es erlaubt solche Änderungen einfach und effizient umzusetzen. IT Systeme werden mit sogenannten "Workflow Systemen" integriert, welche Geschäftsprozesse in Form einer "Orchestrierung" von Anwendungslogik ausführen. Solche Prozesse werden zentral ausgeführt, die Workflow Engine ist eine zentrale Einheit die die Ausführung einzelner Aktivitäten eines Prozesses steuert. Diese Art der zentralen Ausführung eines Geschäftsprozesses hat signifikante Nachteile, die zum Vorschein treten wenn Änderungen wie beispielsweise das Outsourcing von Teilbereichen eines Geschäftszweiges, implementiert werden. Mit Workflow-Engines der aktuellen Generation führt dies zur Entwicklung von Prozesspartitionen die separat gewartet werden müssen und nicht entstanden sind weil sich das ursprüngliche Geschäftsziel geändert hat, sondern weil aktuelle Workflow-Engines es nicht erlauben einen Prozess durch flexibles Deployment an sich ändernde IT Infrastrukturen anzupassen.

Diese Arbeit schlägt ein fundamental neues Ausführungsmodell für die dezentralisierte Ausführung von Geschäftsprozessen vor, um das zuvor beschriebene Problem zu lösen. Das vorgeschlagene Modell erlaubt das Deployment von beliebigen Partitionen des ursprünglich zentral ausgeführten Prozessmodells, die jeweils auf die sich ändernde IT Infrastruktur angepasst sind. Damit werden Änderungen an dem ursprünglichen Prozessmodell vermieden, die ausschließlich durch Änderungen an der darunterliegenden IT Infrastruktur und nicht durch Änderungen des eigentlichen Geschäftsprozesses motiviert sind.

Die wissenschaftlichen Beiträge dieser Arbeit sind im Einzelnen: (i) Eine detaillierte Analyse der Unterschiede von Tuplespace-basierten und Message-orientierten Middleware Systemen, (ii) eine Petrinetz basierte Formalisierung von Interaktionen mit Tuplespaces, später "Executable Workflow Nets" (EWFN) genannt, (iii) eine vollständige Formalisierung von WS-BPEL 2.0 mit EWFNs, (iv) Algorithmen zur Ausführung von Petrinetzen mit Tuplespaces und (v) die Architektur und Implementierung des "Koordinationskerns" einer dezentralisierten Workflow Engine, die EWFN basierte Workflows ausführt.

Grundlagen und verwandte Arbeiten

Die dezentrale Ausführung von Prozessen ist ein weites Gebiet in der aktuellen Forschung. In Kapitel 2 werden deshalb grundlegende Technologien wie Messaging, Tuplespaces, Web Services und SOA sowie WS-BPEL vorgestellt. Anschließend werden verwandte Arbeiten, unterteilt in die Bereiche Coordination Languages, Petrinetze im Workflow Management und verteilte Prozessausführung, besprochen.

Das Hauptunterscheidungsmerkmal dieser Arbeit im Vergleich zu verwandten Arbeiten ist die Vermeidung eines Punktes mit zentraler Kontrolle über den Prozess. Dies wird durch die Abbildung der Semantik von Prozessmodellen, die mit Hochsprachen wie zum Beispiel WS-BPEL definiert wurden, auf einen Mechanismus der auf der Weitergabe von Marken basiert, erreicht. Folgende Eigenschaften dieses Systems unterscheiden es von verwandten Ansätzen: (i) Es erlaubt die beliebige Verteilung von Aktivitäten eines Prozessmodells. (ii) Es unterstützt eine automatische, halb automatische und vollständig benutzerge-

steuerte Verteilung eines gegebenen Prozessmodells und erlaubt dadurch, (iii) Partitionen eines Prozessmodells zu bilden, die sowohl durch Geschäftsziele wie beispielsweise Outsourcing, als auch durch technische Gründe wie beispielsweise die Verbesserung der Ausführungszeit oder des Durchsatzes beeinflusst werden. Des Weiteren bietet der vorgestellte Ansatz die (iv) vollständige Unterstützung von WS-BPEL 2.0, nicht nur einer eingeschränkten Untermenge der Sprache und ist (v) unabhängig von einer konkreten Prozessdefinitionssprache.

Vergleich von Message-orientierter und Tuplespace-basierter Middleware

Middlewaresysteme spielen eine fundamentale Rolle beim Entwurf einer Workflow Engine. In Kapitel 3 wird deshalb die Rolle von Middleware in Workflow Systemen im Allgemeinen untersucht. Anschließend fokussiert sich die Diskussion auf Message-orientierte Middleware (MOM) und deren Rolle zur Umsetzung des "Navigators", der zentralen Komponente jedes Workflow Management Systems. Im Anschluss wird Message-orientierte Middleware mit Tuplespaces, einer sehr ähnlichen Technologie verglichen. Der Vergleich findet in zwei Phasen statt: In der ersten Phase werden beide Systeme auf Basis ihrer Eigenschaften, unabhängig eines konkreten Einsatzgebietes verglichen. Im zweiten Teil bilden Enterprise Integration Patterns die Basis des Vergleiches, indem Tuplespaces anstelle von MOM als Implementierungsgrundlage angenommen wird.

Tuplespaces eignen sich im Besonderen für den Ansatz der verteilten Prozessausführung: Ähnlich zu dem vorgestellten Prozessausführungsmodell kommunizieren Tuplespace-basierte Programme durch lesen und schreiben von Tuples[1] in einen gemeinsamen Speicher. Außerdem bieten Tuplespaces Mechanismen, die es einzelnen Tuplespace Clients erlaubt sich untereinander zu koordinieren, um beispielsweise die Abfolge einzelner Schritte die zum Erreichen eines gemeinsamen Ziels nötig sind, untereinander abzustimmen.

[1] Tuples werden hier als eine Generalisierung von Marken verstanden, in Anlehnung an den Begriff des Tokens in Stellen-Transitions-, und gefärbten Petrinetzen.

EWFN - Ein Modell zur verteilten Prozessausführung

Im Anschluss wird in Kapitel 4 ein Modell zur formalen Beschreibung von Interaktionen mit Tuplespaces entwickelt. Es basiert auf der Ähnlichkeit von Petrinetzen und Tuplespaces und baut deshalb auf nicht hierarchischen, gefärbten Petrinetzen auf, die mit Lesekanten und einer an Linda Templates angelehnten Template Sprache [Gel85] anstelle von Standard ML erweitert wurden. Die entstandenen Netze werden "Executable Workflow Nets" (EWFN) genannt. Jedes Element des Modells hat ein direktes Äquivalent in einer Tuplespace-basierten Applikation, so dass EWFNs *direkt* mit Tuplespace Middleware ausgeführt werden können. Es wird die formale Beschreibung von EWFNs sowie deren Semantik besprochen und überdies gezeigt, wie EWFNs auf Stellen-Transitionsnetze abgebildet werden können, um beispielsweise Standard Petrinetz Analysetechniken auch auf EWFNs anwenden zu können.

Transformation von WS-BPEL 2.0 nach EWFN

EWFNs sind eine formale Beschreibung für Tuplespace-basierte Anwendungen. Um Prozessmodelle damit ausführen zu können ist eine Abbildung einer Prozessdefinitionssprache, wie zum Beispiel WS-BPEL, nach EWFN erforderlich. In Kapitel 5 wird eine solche Abbildung beschrieben, in dem die Semantik jeder WS-BPEL 2.0 Aktivität als EWFN Graph dargestellt wird. Damit wird einem bewährten Vorgehen bei Workflow Systemen gefolgt: Vor der eigentlichen Ausführung eines Prozesses übersetzt auch Apache ODE zum Beispiel jeden BPEL Prozess in ein internes Ausführungsformat.

Tuplespace-basierte Ausführung von EWFNs

Im Anschluss wird in Kapitel 6 die Ausführung von EWFNs mithilfe von Tuplespaces besprochen. Das Kapitel beginnt mit einer generischen Betrachtung der Ausführung von Petrinetzen mit Tuplespaces und es wird gezeigt, wie Stellen-Transitionsnetze und gefärbte Petrinetze effizient mit Tuplespaces ausgeführt werden können. Dabei werden auch entsprechende Algorithmen

vorgestellt, die teilweise auf erweiterten Tuplespace Operationen wie zum Beispiel Multimengen Operationen basieren. Des Weiteren wird das Problem der Kontrollflusssynchronisation in Tuplespaces diskutiert und die `sync` Operation als eine Lösung vorgestellt.

Architektur und Implementierung

Das nächste Kapitel dieser Arbeit beschreibt die Architektur und Implementierung eines Tuplespace Kernels der EWFNs "nativ" ausführen kann und damit die Basis der dezentralen Workflow Engine bildet. Dieser Tuplespace Kernel implementiert die Operationen zur effizienten Ausführung von EWFNs, welche im vorhergehenden Kapitel besprochen wurden. Des Weiteren definiert dieses Kapitel EWFN-ML, ein PNML-basiertes Serialisierungsformat für EWFNs.

Im letzten Kapitel werden die wissenschaftlichen Beiträge dieser Arbeit zusammengefasst und Bereiche für weitergehende Arbeiten diskutiert.

Abstract

The ability to cope with change is one of the biggest challenges companies are facing today [JLN08]. Methods to increase business agility, such as outsourcing or off-shoring of non-core parts of the business are practiced already and will continue to play an important role in the future. A change in business however always means to also change the underlying IT infrastructure, which therefore needs to be designed in a way to be able to support flexible re-arrangement of its support for business operations. IT systems today are integrated using so called workflow management systems, that execute a business process in the form of "orchestrations" of the individual applications used for carrying out business functions.

These processes today are enacted centrally, i.e. the workflow engine acts as a central point of control governing the lifecycle of all activities of a process. This way of enacting processes has certain drawbacks, revealed when executing change such as outsourcing parts of a process. Using state-of-the-art workflow engines, this results in development of process partitions that need to be maintained separately and were not created because the original business goal has changed, but because the used workflow engine does not allow for flexible deployment of a single process model to multiple process engines.

This thesis proposes a fundamentally new execution model for decentralized workflow enactment that does not have these drawbacks. It allows for flexible

deployment of arbitrary partitions of the original workflow graph, thus supporting change at the level of business processes avoiding the need to change a process solely because the underlying IT infrastructure has changed.

The contributions of this thesis are: an in-depth analysis of differences between tuplespace-based and message-oriented middleware systems, a Petri net based formalization of tuplespace interactions called Executable Workflow Nets (EWFN), a fully-fledged Petri net formalization of WS-BPEL 2.0, a method and algorithm to execute Petri nets using tuplespaces and an architecture and implementation of the "coordination kernel" of a decentralized workflow engine that executes Petri net based workflows.

In the following, we present a summary of each chapter of this thesis.

Background and Related Work

Distributed Workflow enactment is a wide field in today's research on business process management. Chapter 2 consequently provides background on prominent technologies and surveys the field by highlighting key contributions. Relevant publications are analyzed with a focus on coordination languages, the use of Petri nets in workflow management and selected approaches in distributed workflow enactment.

The main differentiator of our work compared to state of the art is the avoidance of a central point of control by mapping the semantics of a given process execution language to token passing and thus allowing a given process model to be arbitrarily distributed. This exhibits the following characteristics that distinguish our approach from existing work: (i) arbitrary distribution of activities of a given process model, (ii) support for automatic, semi-automatic and fully user-driven partitioning of a process, and thereby, (iii) the ability to create partitions of a process influenced by business reasons e.g. outsourcing, as well as technical reasons such as better execution time or throughput, (iv) full-fledged support for WS-BPEL 2.0, not only a restricted subset of the language, (v) independence of a certain process definition language.

Comparison of Message-Oriented and Tuplespace-Based Middleware

A prerequisite to the decentralized execution model is a clear understanding of the concepts behind Linda [Gel85] and tuplespace computing. Before designing the execution model, in Chapter 3 we therefore analyze the properties of tuplespaces and compare them to message-oriented middleware, a technology found to be very similar by related work. While the comparisons done before use high-level properties such as levels of decoupling [AvdADtH05] or simple message exchange patterns [FKLT07, Ley06], the basis of our comparison are Enterprise Integration Patterns [HWB03]. These patterns represent harvested knowledge from almost two decades of applied use of message-oriented middleware in enterprise application integration (EAI) scenarios. For comparison, we assume tuplespaces instead of message-oriented middleware as the underpinning of the patterns and in that way, reveal the differences between both technologies.

The knowledge gained from the comparison is then used to decide whether tuplespaces or message-oriented middleware is a suitable basis to implement the inherent requirements such as absence of a central point of control or certain quality of services (QoS) to build a decentralized workflow engine. We conclude that tuplespaces are superior to messaging systems for our application. The drivers for this decision are that the tuplespace coordination model is very similar to Petri nets and thus to the execution model we foresee. Tuplespaces do not enforce a direction in communication and allow random access to tuples in a tuplespace. Furthermore, tuplespaces allow for destructive as well as non-destructive retrieval of tuples, enabling us to use the same middleware to handle both: data access (typically handled using non-destructive read operations) and control flow (typically expressed using destructive read operations). Additionally, tuplespaces implement the demanded decoupling dimensions and do not require a central point of control, the key requirement for independent activities and thus independent process partitions.

EWFN - An Execution Model for Decentralized Workflow Enactment

Next, a formal model to describe tuplespace-based interactions is developed in Chapter 4. Based on the close relationship of Petri nets and tuplespaces, this model uses non-hierarchical, Colored Petri Nets extended with read arcs and a Linda like template language [Gel85] instead of CPN ML for arc inscriptions. The resulting Petri net dialect is called "Executable Workflow Nets" (EWFN), as each element of the model has an equivalent element in a tuplespace-based application, making EWFNs *directly* executable on tuplespace-based middleware. We provide a formal description of the syntax and semantics of EWFNs, as well as a mapping to Place-Transition Nets, allowing standard Petri net analysis and verification techniques to be used with EWFNs.

Transformation of WS-BPEL 2.0 to EWFN

The EWFN model allows to describe tuplespace-based applications in a formal way. In order to execute process models using tuplespaces, a mapping from a process definition language such as WS-BPEL is required. In Chapter 5 we therefore express each element of the WS-BPEL 2.0 specification in form of an EWFN graph, specifying the semantics of a BPEL element by tuplespace operations and thus in a way that is executable directly on tuplespace systems. By doing so, we follow common practice in the implementation of workflow systems: Apache ODE[1] for instance translates BPEL constructs into an internal object model that is executed using Jacob, a framework that provides – very much like tuplespaces do in our case – support for persistence and concurrency and thus builds the basis for process execution[2].

[1] `http://ode.apache.org/`

[2] Please note that we use footnotes for webpages only. Literature found on the web will be referenced in the bibliography.

Tuplespace-Based Execution of EWFNs

Subsequently, Chapter 6 discusses how to execute EWFNs using tuplespaces. We start by introducing several possibilities of execution based on different kinds of Petri nets such as Place-Transition or Colored Petri Nets. A major aspect of executing Petri nets on tuplespaces is the way how the structure of the Petri net graph is encoded in tuples. For efficient execution, we present algorithms to implement basic tuplespace operations such as writing, destructive and non-destructive consumption of tuples as well as for batch operations such as multi-read or update. Furthermore, the problem of synchronized consumption of tuples in tuplespaces is discussed and a solution in form of the `sync` operation is proposed.

Architecture and Implementation

The next part of this thesis describes the architecture and implementation of the tuplespace kernel that natively executes EWFNs and thus builds the basis for the implemented, decentralized workflow engine. In particular, the presented tuplespace kernel implements the tuplespace operations discussed in the previous chapter for efficient EWFN execution. This chapter also defines EWFN-ML, the XML-based serialization format for EWFNs.

Finally, in Chapter 8 a summary of the contributions is presented and several areas for future research are identified.

Acknowledgments

Although written by just one, this is a work of many people. I am deeply grateful to all who helped making this dissertation a reality.

Prof. Dr. Frank Leymann for his confidence and trust in me when I was working towards being accepted as a PhD candidate, for his guidance and support during all the time that followed, as well as for his insight, time and effort for supervising this thesis.

Prof. Dr. Eva Kühn for starting the discussion about tuplespaces and workflow systems and for igniting the idea of a "process space" during her visit to the IAAS in the summer 2006. This discussion actually marked the beginning of two PhD projects.

All my co-workers at the IAAS for being fabulous colleagues and friends and for throwing so many nice parties and spontaneous get-togethers. Especially Daniel Wutke, my office-mate and the other member of the process space team for the countless and very inspiring discussions on all aspects of the thesis. The "LaTeX competence team" Daniel Wutke, Tammo van Lessen and Oliver Kopp for help on all matters of typesetting with LaTeX and for the team effort of creating our beautiful LaTeX thesis template. I would also like to thank Oliver Kopp and

Tobias Unger for their critical proof reading of parts of this thesis and Dimka Karastoyanova for her advice on thesis writing in general.

My family for their support through the entire time and for the excellent food when I was staying for the weekend.

Sylvia Bopp for proof reading, patience and love. And also for those many delicious meals that allowed me to escape from the university cafeteria.

Finally, I would like to thank all the people I may have forgotten, the staff at my local gym for letting me work-out late, the TeXlipse team for creating the best LaTeX editor ever and the last.fm breakbeat radio channel for always playing the right kind of music.

This has been a wonderful experience. I am very much looking forward to what the future will bring next.

Chapter 1

Introduction

Companies today find themselves in a very competitive and extremely fast changing environment, influenced by the effects of globalization such as mergers and acquisitions and the ever increasing pace of advances in technology. Clearly, the ability to change and adapt to new situations is a key competitive advantage to stay in and atop of business. Or, as put by "Making Change Work", a study developed by IBM's Global Business Services Strategy & Change Practice:

> *For its very survival, the Enterprise of the Future must better prepare itself as the pace, variety and pervasiveness of change continue to increase. [JLN08]*

Outsourcing non-core business processes, such as payroll, human resource administration, or accounts payable and receivable is a prominent solution to tackle these challenges. By outsourcing, the then leaner and more focused company improves its agility and thus is able to react more quickly to changing

conditions in today's markets.

Outsourcing often splits business processes apart into multiple pieces possibly distributing them over multiple locations. Yet, it is expected that a company presents itself as a coherent whole that, although operating globally, is perceived as being tightly integrated.

SOA and Web service technologies embody a considerable step towards providing this kind of agile, yet integrated style to the IT-level of businesses and through that to the entire business itself: standards for the definition of interfaces, messages and protocols ensure interoperability between software products, significantly reducing the effort to integrate or add new systems to existing IT landscapes.

Additionally, Web services are designed to be composable, allowing to create new, coarser-grained services in form of an *orchestration* of existing services provided by enterprise applications. This way of creating new services by means of composition is an implementation of the two-level-programming paradigm [LR97, WCL+05], enforcing clear separation between process and application logic. Explicit separation of process logic from the applications providing individual business functions is a core requirement for IT system agility, leading to significantly faster adaptation times of IT systems to changes demanded from the business[1].

Such processes are often specified in WS-BPEL [A+07] (or BPEL for short), a Web service standard specifically designed to represent business processes in form of Web service orchestrations. A process today is enacted (logically) centrally, i.e. a single component (called "navigator") which is part of a process engine governs the activities to be executed. Even in the case of a clustered engine, every process instance is navigated "centrally", i.e. by exactly one node at a time within the cluster. This central way of executing a business process is problematic in cases when organizational changes require to outsource a subset of a process' activities or when unnecessary remote communication for reasons of performance optimization needs to be avoided. Such changes

[1] Please note that in this thesis we use the terms "workflow" and "business process" interchangeably. Similarly, we treat "process" and "process model" as synonyms when it is clear from the context whether the model instance or the model itself is meant.

require a single process model to be executed by multiple process engines, but still leave the operational semantics of the original process model intact. These cases today require re-design of the original process and consequently re-deployment and maintenance of customized process models representing partitions of the original process logic. Moreover, these customized models were not designed with the business goal of the original process model in mind; instead, their design was driven by limitations of the process engine that only allows to execute a process model as a whole.

In this thesis, we address these problems by proposing a fundamentally new way of executing processes that does not rely on a central navigator component. Instead, independent, distributed navigation components responsible for parts of the process *coordinate themselves* in a peer-to-peer manner in order to maintain the operational semantics of the original process. As a result, processes are invariant to changes concerning re-location of parts of the orchestration logic. The same process can be executed centralized, as well as in various distribution configurations, executing orchestration logic in exactly the location required – without any modification of the underlying process model. The novel architecture covers the full spectrum of execution scenarios ranging from traditional, centralized process execution over arbitrarily partitioned process models to fully decentralized (i.e. "discrete") process execution where each activity resides at a different location. It is important to note however that our approach is not limited to BPEL processes; the *coordination model* developed in this thesis was designed to be able to enact processes specified in different process definition languages such as BPEL or BPMN [Wes07, DvdAtH05].

In the following, three different scenarios emphasize the advantages of the new architecture, allowing for arbitrary distribution of orchestration logic while maintaining the operational semantics of the original process definition.

1.1 Scenario 1: Organizational Change

Assume the process model as shown at the left-hand side in Figure 1.1 is run by a single department, i.e. all activities (A through E) and their implementations depicted as Web services are under the control and thus e.g. run on a machine

hosted by that department. Changes in business require that parts of the process – and thus parts of the work the department is doing – are subcontracted to a third party.

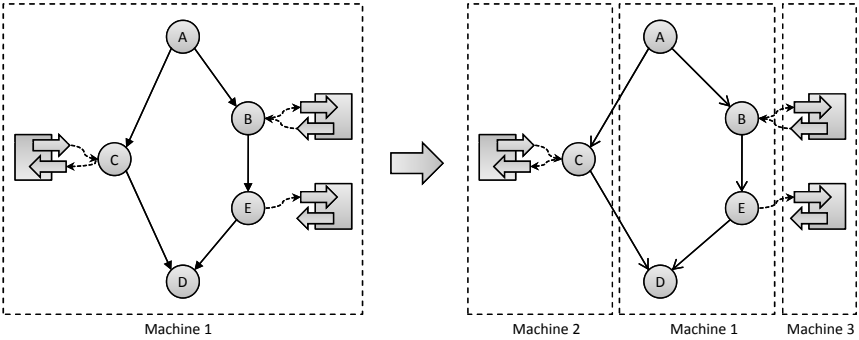


Figure 1.1: Scenario 1: Organizational Change

The resulting process is depicted at the right-hand side of Figure 1.1. The process model itself has not changed – the business goal and the individual steps carried out are still the same. What has changed however are the responsibilities and thus also the actual IT systems involved in the process. Machine 1 that was executing the entire process before still executes most parts of the process; activity *C* however is subcontracted to another company that uses its own business applications to carry out the work. Consequently, the activity and its implementation (in form of a Web service) are now running on the premises of the subcontractor (machine 2).

Another required change was to outsource the applications implementing activities *B* and *E*, possibly to a SaaS provider. The activity implementations are therefore running on a third machine instead of machine 1 as in the original process model.

1.2 Scenario 2: Infrastructural Change

A company runs its processes using a workflow engine that is hosted by one machine, possibly a powerful mainframe computer. For cost reasons, it is decided that this machine is replaced by a cluster of smaller machines, e.g. built using cheap commodity hardware. Using our model, the same business process can be executed on the set of machines as if it still was executed by the original, single machine. The semantics of the original process model is preserved independent from its deployment.

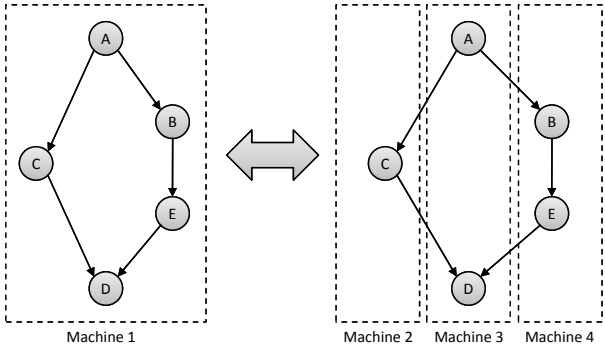


Figure 1.2: Scenario 2: Infrastructural Change

The key enabler of this scenario is the ability of the proposed execution model to preserve the process's operational semantics over different distributed deployments. This is depicted by Figure 1.2: it shows the same process model (without showing activity implementations in form of Web services for simplicity) in a centralized, as well as in a distributed setting. However – as indicated by the double arrow – it is also possible to deploy a decentralized process in a centralized manner.

The flexibility to deploy process models to different configurations independent of the underlying IT infrastructure allows for consolidation of IT infrastructures without impact on the operational semantics of the process to be executed.

1.3 Scenario 3: Optimized Process Execution

In this scenario (which is similar to the one presented in [Wut10]), the motivation for changing the deployment of a given process model is to achieve more efficient execution. Assume a poor network connection between machine 1 and machine 2 on the left hand side of Figure 1.3, possibly because machine 2 is located in a remote office of the company running the process. Furthermore, large messages are exchanged between process activities B and E and their Web service implementations, indicated by slightly thicker arrows in Figure 1.3.

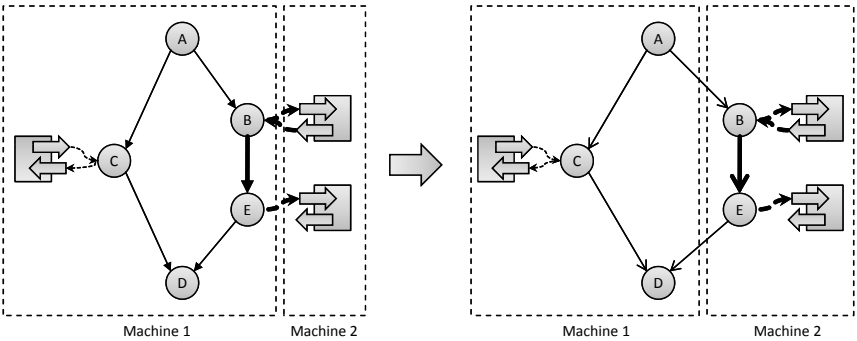


Figure 1.3: Scenario 3: Optimized Process Execution

It is much more efficient to change the deployment of the process to the scenario depicted on the right-hand side of Figure 1.3. In this situation, the transfer of large amounts of data does not cross system boundaries and does not require network communication as in the initial deployment. Instead, only control flow is passed between machine 1 and machine 2; the problematic data flow is reduced to purely local interactions, which is much more efficient than the remote communication that was required in the initial situation.

The key benefit in this scenario is that our workflow system allows to provision process models to the target IT infrastructure in such ways that the given resources are optimally used.

In the presented scenarios, it is not the process language that prevents

change; it is the underlying execution engine that needs to support flexible deployments of process models to distributed and diverse IT infrastructures. This work addresses the need for such a distributed process engine, and proposes a tuplespace-based[1] coordination model as the key mechanism to allow for decentralized execution of a given process. The proposed coordination model is independent of a specific process definition language. BPEL was chosen since it is a widely used and an industry accepted standard for executable process definitions.

1.4 Focus of This Work

Figure 1.4 presents an overview of the main steps (boxes) and documents (boxes with wavy bottom side) involved when using our decentralized workflow engine. We start with a BPEL process model that is transformed to an internal format called *Executable Workflow Network Markup Language* (EWFN-ML) used as input for the partitioning process, which creates a description of the mapping of process elements to parts of the underlying IT infrastructure in form of a Distributed Deployment Descriptor (DDD). Next, the original BPEL file and the Distributed Deployment Descriptor are used for deployment and finally decentralized execution of the process using a novel, tuplespace-based process execution middleware called "Process Space".

As indicated by the dotted boxes, this work is centered on the decentralized execution model called EWFN, which is the theoretical underpinning of our new process engine. In this work, we cover the first three steps of the figure: besides developing the actual execution model and evaluating the underlying middleware, we develop a transformation from BPEL into the new execution format. Additionally, a serialization for EWFNs, the EWFN Markup Language (EWFN-ML), is defined and the tuplespace-kernel of the execution middleware is developed.

[Wut10] builds upon this work and describes the overall system architec-

[1] In this thesis, the word *tuplespace* is used to denote the class of middleware systems that arose from the original Linda concept [Gel85]. In particular, this includes systems that build upon Linda, e.g. through the introduction of transactions, multiset operations, leases, etc.

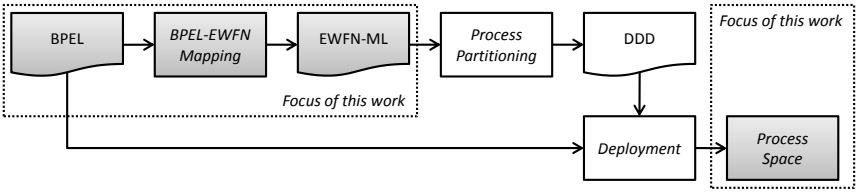


Figure 1.4: Focus of this Work

ture including algorithms for partitioning the process model [WML09a], functionality for process deployment using the distributed deployment descriptor [WML09b], transactionality, logging and auditing [WML09b, WML08b].

1.5 Summary of Scientific Contributions

This thesis contains five key contributions as outlined below:

Contribution 1: in-depth analysis of differentiators between tuplespaces and MOM

Since we are proposing a novel workflow execution architecture, which today is often based on message-oriented middleware (MOM) systems, we first investigate their differences to tuplespaces. State of the art middleware research analyzes the differences of tuplespaces and message-oriented middleware based on high-level decoupling dimensions like time, reference or location [AvdADtH05, Ley06] or uses simple communication patterns [FKLT07]. In this thesis, we use a fundamentally different approach to achieve this goal: our basis are Enterprise Integration Patterns [HWB03], that capture more than a decade of best practices in application of message-oriented middleware to enterprise integration scenarios. As these patterns propose solutions to recurring problems in the domain of EAI, which is the primary application area of MOM, we use them as the basis for our comparison [MWSL07]. As a result, we find more fine-grained distinctions between MOM and tuplespaces than have been described in the literature

before. Based on these findings, we analyze the suitability of tuplespace middleware as basis for building a decentralized workflow engine. We argue that tuplespaces are ideal since they combine basic features of databases and messaging systems, two of the most prominent kinds of middleware underlying today's workflow systems.

Contribution 2: distributed process execution model

In order to provide a formal basis for distributed process execution, we develop an extension of Petri nets called Executable Workflow Nets (EWFN) [MWL08a, MWL08b], which form the basis for the distributed process execution model proposed in this thesis. EWFNs fill two important gaps in state-of-the-art research: (i) they present a fundamentally new way to execute workflows by decentralized *coordination* of fully decoupled, individual navigation components instead of employing central *navigation*. This allows for arbitrary deployment of partitions of the workflow graph leading to greater flexibility of process model deployments. Additionally (ii), it defines a formal model and a graphical language for the expression of tuplespace-based interactions.

Contribution 3: transformation of WS-BPEL 2.0 to EWFNs

Today, BPEL processes are enacted centrally – there is one single standalone component that governs the lifecycle of all activities the process is composed of. We address this shortcoming by providing a complete mapping from BPEL 2.0 to EWFNs, applying our model to an industry accepted standard and thus enabling BPEL processes to be enacted in a decentralized manner. The semantics of each BPEL activity and the data and control flow implied, are expressed by means of sending and retrieving tokens to places. The explicit availability of control flow and especially data flow in the execution model is a requirement for the graph partitioning step that is later applied to the EWFN [Wut10] as the creation of individual partitions from the original process model is heavily influenced by the amount of data shipped between activities.

The EWFN formalization of BPEL [WML09a] in this work is a fully-fledged Petri net formalization of BPEL 2.0, covering in particular advanced concepts such as Dead-Path-Elimination, event-, fault-, and compensation handling and correlation. Furthermore – as the result of being executable – data flow as well as control flow are modeled explicitly, and thus are an integral part of the workflow model. This is an important differentiator to related approaches such as [Loh08] or [OVvdA+05a], which are targeted towards verification of BPEL rather than its execution.

Contribution 4: a method to execute Petri nets using tuplespaces

The close relationship between Petri nets and tuplespaces has been recognized before [BR06] and was even mentioned by Carl Adam Petri himself, yet it is not further investigated nor exploited in current research. In contrast, in our work this close relationship forms the basis of the new workflow engine architecture we propose. In form of EWFNs, Petri nets are used to formalize and execute tuplespace-based interactions. Petri nets provide the formal underpinning of our workflow engine and the tuplespace middleware provides necessary QoS such as reliability, persistence and transactionality for execution.

Consequently, in [MWL08d, MWL09] we introduce a generic method to execute Petri nets using tuplespaces and evaluate different tuple encodings of Petri net graphs. As a result, we propose to use specific encodings depending of the kind of Petri net to be executed. Based on these generic results that apply to different classes of Petri nets, a method to execute EWFNs is derived.

Synchronized consumption of multiple tuples with additional requirements such as partially equal fields (e.g. to consume all tuples which belong to the same process instance at once) is not well supported in current tuplespace systems. Cumbersome programming using client-side logic that groups multiple tuplespace operations in a transaction is required to support such cases. Moreover, synchronized consumption of tuples from multiple tuplespaces requires a distributed transaction to be run between

participating tuplespaces. The described scenario however is essential when implementing control flow and specifically synchronization of multiple parallel threads of control flow in a tuplespace-based workflow system.

As a result, we provide an algorithm for efficient (cf. Chapter 7 for details), synchronized consumption of tuples [MWL08c] that allows to express constraints – such as partially equal fields – between matching tuples. An additional operation on the coordination interface of a tuplespace that implements our algorithm is introduced. To solve the case of synchronized consumption of tuples from multiple tuplespaces, we present a pattern to reduce the problem to the non-distributed case, which is solved by the proposed algorithm.

Contribution 5: architecture and implementation of a tuplespace kernel for efficient EWFN execution

Tuplespace-based, decentralized execution of workflow definitions is a new approach to distributed workflow enactment. Especially, there is no tuplespace implementation that is tailored specifically to executing decentralized workflows. We present the architecture and implementation [WML08b, WML09b] of our coordination kernel and thus the core of a tuplespace system that supports our algorithm and pattern for efficient synchronization of control flow. An evaluation using different encodings of the same Petri net shows the advantages of passing control flow using our coordination primitives compared to JavaSpaces, a state-of-the-art tuplespace implementation.

1.6 Organization

This thesis is organized as follows: Chapter 2 presents background and related work, including discussions on approaches in the field of coordination languages, distributed workflow enactment and the role of Petri nets in workflow systems. Chapter 3 provides an extensive discussion on the differences of message-oriented middleware and tuplespace systems and motivates the use of

tuplespaces for distributed workflow enactment. In Chapter 4, we build the groundwork for the distributed workflow engine by providing a Petri net based formalization of tuplespace-based interactions called Executable Workflow Nets (EWFN) – the workflow engines' "execution model". This work is then used in Chapter 5, to express WS-BPEL 2.0, an established workflow definition language, in the new execution model. Subsequently, Chapter 6 provides a generic discussion on how to execute Petri nets on tuplespaces, and then focuses on how to efficiently execute EWFNs. Next, Chapter 7 describes the architecture of the tuplespace-based coordination kernel that forms the basis of the decentralized workflow engine. Finally, we conclude the thesis in Chapter 8 with a summary of the main contribution of this work and a description of open research areas.

Chapter 2

Background and Related Work

This chapter is structured into two parts: Section 2.1 provides a brief overview of relevant technologies needed to understand the contributions of this work. Section 2.2 places this work into context of existing contributions, outlines the novelties of our approach and highlights differences to related approaches.

2.1 Background

This section provides a summary on background technologies this work uses and builds upon. We briefly introduce message-oriented middleware (MOM) and the landscape of patterns documenting best-practices around their use, introduce tuplespace systems and Petri nets and summarize the key concepts behind SOA and Web services. As part of the Web service technology stack, we highlight WS-BPEL, the service orchestration standard we use to define process models. If required, additional details about a particular technology are provided directly in the chapters that deal with the subject-matter.

2.1.1 Message-Oriented Middleware

The concept behind messaging, i.e. asynchronous communication through buffered messages, existed long before the terms "messaging" and "message-oriented middleware" were introduced. It existed in many different forms, hidden as libraries in operating systems, application products etc. and was not available in a generic form, e.g. as a separate product [BHR95]. The actual beginnings of messaging can be traced back to transaction monitor-based systems such as CICS™ or IMS™ [BN97] that control sequences of messages and group them into transactions. IMS pioneered the *queued transactions* approach where multiple messages contain a transaction identifier, the name of the program to be executed and all necessary input data. Such messages are put into a *queue* and processed asynchronously by the services provided by the IMS subsystem. In IMS, *message queues* are the interfaces between an application program and IMS services.

Message-oriented middleware relies on a few core concepts that are detailed below:

Message A *message* in message-oriented middleware defines a "self-contained business request", i.e. it contains all data (e.g. input data, reply address, correlation information, etc.) required to perform a certain request. A message is structured into a *header* and a *body* part. The message header contains information directed at intermediary nodes the message passes through during delivery. The message body is directed at the ultimate receiver of a message.

Asynchronous communication Communication using a messaging system is inherently asynchronous: once a message is accepted by the system, the sender is un-blocked and may perform other work while the message is being processed (a.k.a. *send and forget* behavior). The message itself is buffered by the messaging system.

Loose coupling Parties communicating over a messaging system are "loosely coupled", very few assumptions must be made about each other when ex-

changing information. Messaging provides autonomy in the dimensions *time*, *location* and *reference* [AvdADtH05].

Guaranteed delivery A message accepted for delivery by the messaging system will eventually be delivered. A message must not be lost. Unroutable messages (e.g. a message forwarded to a response queue no longer existing) are stored in *dead letter queues*. Unprocessable messages (e.g. a client rejects a message because of data format errors) are stored in an *invalid message queue*. A messaging system employs *store-and-forward* alike techniques to ensure that messages are not lost during transmission between individual MOMs.

Messaging middleware is primarily used as communication infrastructure to integrate business applications, a field generally referred to as enterprise application integration (EAI). There exist a large number of patterns that capture recurring problems and corresponding solutions using messaging in the EAI domain [HWB03]. Many commercial messaging middleware products are available in the market today; amongst others, these are IBM WebSphere MQ[1], Microsoft Message Queuing[2] (MSMQ) or JBoss Messaging[3].

Messaging middleware is often used to implement the communication infrastructure backbone providing reliable transport for Enterprise Service Bus [Cha04, Ley05] (ESB) implementations. Prominent open-source examples are Apache Synapse[4] or Mule ESB[5].

In Chapter 3, we investigate the role of messaging systems in workflow engine architectures, especially how they are used to implement the navigator component. We then compare messaging to tuplespaces, our middleware system of choice as the underpinning for the implementation of a decentralized navigator.

[1] `http://www-01.ibm.com/software/integration/wmq/`
[2] `http://www.microsoft.com/windowsserver2003/technologies/msmq/`
[3] `http://www.jboss.org/jbossmessaging/`
[4] `http://synapse.apache.org/`
[5] `http://www.mulesource.org/`

2.1.2 Tuplespaces

Tuplespace technology has its origin in the Linda coordination language, defined by [Gel85] as a parallel programming extension to the "C" and Fortran programming languages for the purpose of separating coordination logic from program logic [GC92]. The Linda concept is built around the notion of a *tuplespace* – associative memory that is shared among all interacting parties. A user interacts with the tuplespace by storing and retrieving *tuples* (i.e. an ordered list of typed fields) via a simple interface: tuples can be (i) stored using the `write` operation (originally called `out`), (ii) retrieved destructively using `take` (originally called `in`) and (iii) retrieved non-destructively using the `read` operation (`rd` in the original Linda model). Tuples to be read or taken are identified using a template mechanism, by providing values of a subset of the typed fields of the tuple to be read (a.k.a. "associative addressing"), similar to *query by example* [Zlo77]. Using tuplespace-based coordination, execution of a component's computational logic is triggered when tuples matching the templates registered by the respective component are present in the tuplespace – a tuplespace operation blocks execution of the current thread until a matching tuple can be found.

The original tuplespace concept has seen a large amount of modifications since it was created. These include: (i) the introduction of new (higher-level) operations such as notifications or bulk operations, (ii) more expressive matching e.g. through support for predicates, `and`/`or` queries or polymorphic matching, (iii) leases on tuples and (iv) timeouts.

Prominent tuplespace implementations supporting all or many of these extensions are GigaSpaces[1], IBMs TSpaces[2] or Suns JavaSpaces[3]. Recently, the tuplespace paradigm has re-gained attention in research in the area of middleware for the semantic web. Examples are sTuples [KLF04], TripCom [MdFK+08] or TSC [RMRD+06].

Two chapters in this work are centered on tuplespaces: Chapter 3 compares tuplespaces to message-oriented middleware, the predominant middleware in

[1] `http://www.gigaspaces.com/`
[2] `http://www.almaden.ibm.com/cs/TSpaces/`
[3] `http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/`

use today to implement the navigator component of workflow management systems. In Chapter 4 we present EWFNs, a decentralized workflow execution model based on Petri nets using tuplespaces as the underlying execution middleware.

2.1.3 Petri Nets

Petri nets [Rei85] are a formal language with graphical representation to model systems with concurrency and resource sharing. They offer a general theory for discrete and parallel systems and are a generalization of automata theory, adding the concept of concurrently occurring events. Petri nets were originally developed in the 1960s by Carl Adam Petri in his dissertation "Kommunikation mit Automaten" [Pet62]. Since then, Petri nets have become largely used and accepted not only in IT but in many fields where concurrent processes are modeled and analyzed. Examples are the analysis of communication protocols and networks, task planning, manufacturing processes, etc.

Various extensions have been introduced since Petri nets where originally developed. These include extensions introducing time, color (structured tokens) or probabilities. Similarly, restrictions constraining the expressivity but at the same time lower the complexity of algorithms for analysis, have been introduced. These include *Free-Choice* nets with constraints on the number of arcs leaving a place or a transition, or *Marked Graphs*, a class of Petri nets that only allows places to have exactly one incoming and one outgoing arc.

An attempt to classify the many Petri net extensions was made in [BdCdM92], proposing three different Petri net levels[1]:

Level 1 Petri nets are characterized by places which can represent *Boolean values*, i.e. a place is marked by at most one unstructured token. Examples are: Condition-Event nets or 1-Safe nets.

Level 2 Petri nets are characterized by places which can represent *integer values*, i.e. a place is marked by a number of unstructured tokens. An example Petri net type of this class is the Place-Transition net (PT-Net).

[1] Further examples for Petri net dialects categorized by these levels can be found at `http://www.informatik.uni-hamburg.de/TGI/PetriNets/classification/`

Level 3 Petri nets are characterized by places which can represent *high-level values*, i.e. a place is marked by a multiset of *structured tokens*. Colored Petri Nets (CP-Nets) or Predicate-Transition Nets are examples for this level.

Petri nets in level three are also commonly termed "high-level" Petri nets. In this work, colored, non-hierarchical Petri nets build the basis for the formal model of tuplespace interactions called EWFNs, which is developed in Chapter 4.

2.1.4 SOA, Web Services and WS-BPEL

Web service technology [Pap08, Ley03] is one implementation of a service oriented architecture (SOA) [WCL+05]. The main motivation behind its development compared to similar component technologies like CORBA [OH98] are interoperability between platforms, protocols, vendors, etc. through abstractions from technology and provider [B+06].

Figure 2.1 depicts the Web service technology platform as a "stack" of standards, covering layers from *transport* over *messaging* and *description* up to *quality of service* and *components*. In this figure, each layer consists of one or more parts that are implemented by Web service standards which are mentioned in parentheses.

The transport layer is implemented by popular web or enterprise standards such as HTTP, SMTP, etc. It is important to note that Web services are *transport neutral*, custom transports can be implemented by providing an alternative *binding* [C+07a].

The messaging layer contains fundamental XML-based Web service standards such as SOAP as a standardized message format or WS-Addressing for a transport independent way of identifying message sender and receiver. It also contains non-XML based messaging standards such as JMS or AMQP[1] as interoperable ways for message exchanges between Web services.

The description level in the Web services architecture stack consists of standards that define metadata around a service, i.e. the abstract information

[1] `http://www.amqp.org/`

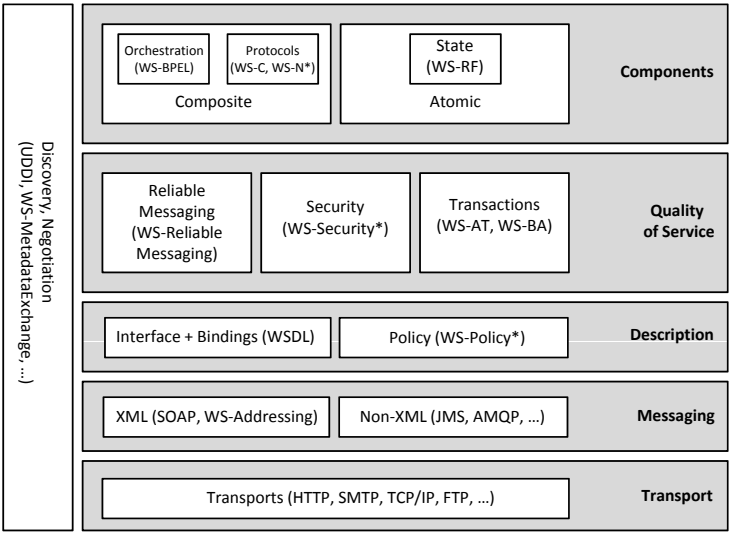


Figure 2.1: Web Services Platform Architecture [WCL+05]

required for discovery and interaction. This layer is dominated by two important standards: WSDL and WS-Policy. WSDL describes the interface and the binding (cf. transport neutrality) of a service whereas WS-Policy defines a way to annotate services with arbitrary metadata describing constraints and conditions in the form of policies, e.g. if it supports transactions, certain QoS, etc. The WS-Policy standard also includes operations on policies, e.g. matching, merging and intersection.

On the quality of service level, standards like WS-ReliableMessaging, WS-Security or WS-BA and WS-AT define interoperable means for reliable, secure and transactional communication between Web services.

The Web service component layer consists of standards describing atomic, stateful services as well as higher-level services in form of service compositions – so called service *orchestrations*. Services that manage explicit state can be described using the WS-RF family of standards. Service orchestrations are defined in the form of business processes using WS-BPEL (or BPEL for short),

one of the key Web service standards used in this thesis.

WS-BPEL [A+07] allows to recursively compose services to new, higher-level services which can in turn be used for further compositions. Using BPEL, services are coordinated by a workflow management system (WfMS) according to a process definition using high-level programming language constructs, so called *activities*. In essence, the process logic expressed in a BPEL file defines when and how services are to be invoked. BPEL offers two kinds of activities: *basic activities* implement functionality for Web service interaction (`invoke`, `receive`, `reply`, `pick`) and data manipulation (`assign`). *Structured activities* in contrast allow for the composition of basic activities and define control flow dependencies between them (`sequence`, `flow`, `if`, etc.). BPEL processes can be modeled either using block-based activities similar to traditional programming languages, using a graph based approach (using the `flow` construct) with nodes and links with transition- and join-conditions between them, or even a mix of both approaches [KMWL08].

2.2 Related Work

Decentralized workflow enactment is a wide field in today's research. In order to show the originality of the individual contributions of this thesis, we structure our analysis of related work into three main parts: Section 2.2.1 covers related approaches in concurrency models and focuses on process calculi, tuplespaces and message-oriented middleware. Since Petri nets play a major role in this thesis, Section 2.2.2 covers research concerning the use of Petri nets in workflow management, particularly for workflow execution, distribution and verification. Finally, Section 2.2.3 surveys related research in distributed workflow enactment and gives details about the execution models used in the area of agent-based systems, Grid or peer-to-peer based approaches and on systems specifically targeting the decentralized execution of BPEL.

2.2.1 Coordination Languages

Coordination languages [OZKT01, PA98] are a subset of the set of formal models for concurrent systems that include CCS [Mil82], CSP [Hoa78] or the π-calculus [Mil99]. They originate from the insight that interactions among software components need to be considered separate from the computation logic of the individual components. Gelernter expressed this finding in the (informal) equation

"Programming = Computation + Coordination" [GC92]

Oftentimes, coordination languages are not designed to be fully fledged programming languages, but rather as extensions (in the form of a set of *coordination primitives*) to traditional programming languages such as C, Java or Ada. The Linda coordination language [Gel85] for instance proposes *tuplespaces* as *coordination medium* and embeds coordination primitives such as `in`, `out`, `rd` or `eval` into C as the "host" programming language. Note that the notion of a coordination language is independent from the concept of a tuplespace. A coordination language defines coordination primitives and their semantics, it does not imply the data structure the primitives operate on [BCGZ01].

Message-oriented middleware (MOM) can be considered another form of coordination middleware, with MQI [BHR95] or JMS [MHC00] as the interfaces describing the coordination primitives and thus the coordination language. Channels (publish-subscribe or point-to-point) are the coordination medium in this case.

In this section, we analyze approaches from the area of coordination languages related to the coordination primitives (and especially the `sync` operation) proposed in Chapter 6. We also review publications that compare messaging to tuplespaces and highlight the novelty of the findings of our own comparison in Chapter 3.

2.2.1.1 Tuplespaces

Tuplespace middleware was invented as an implementation of the notion of the shared data-space the Linda coordination language [Gel85] depends on.

Since their original proposition, tuplespaces where the source of a stream of research with many modifications and extensions to the original tuplespace model, including new coordination primitives, support for transactions, leases, sophisticated tuple matching techniques, etc.

Virtual shared memory [Küh03] is another track of research in the area of tuplespace (or space-based) computing[1]. Here, the focus is more on the shared space and the implementation of its functional and non-functional properties.

Related work in tuplespaces can be roughly divided into two categories based on the kind of coordination primitives employed: *rule-based (or declarative)* tuplespaces propose own rule languages to express coordination, i.e. they allow the tuplespace itself to be programmed (scripting) and thus to create new, higher-level coordination primitives based on existing ones. *Operation-based* tuplespaces instead have a fixed set of coordination primitives which they embed into a host programming language.

Mobile Agent Reactive Spaces (MARS) [CLZ00] and Tuple Centers Spread over Networks (TuCSoN) [OZ98] belong to the class of rule-based tuplespaces. MARS is programmable in Java, TuCSoN employs ReSpecT, a first order logic language defined in [DNO98]. In order to demonstrate the expressiveness of the ReSpecT language, [DNO98] shows how the semantics of Place-Transition Nets (PT-nets) can be implemented using reactions programmed in ReSpecT rules. For that, they use a Petri net encoding technique similar to model 2 (Section 6.1.3) we present in Chapter 6. In follow-up work using ReSpecT [CDRV06], a BPEL orchestration engine based on ReSpecT "Tuple Centres", i.e. ReSpecT-programmable tuplespaces is introduced where BPEL activities are directly expressed as reactions. Interestingly, the authors recognize the need for a graph-based intermediary representation of the process definition, similar to the role EWFNs play in our work.

There are several areas where our approach differs to the work around Tuple Centers and the ReSpecT language: (i) their motivation is a proof-of-concept to show the applicability of tuplespace-based coordination in general and of Tuple Centers in particular to BPEL-based workflow execution. Decentralized work-

[1] `http://www.spacebasedcomputing.org/`

flow execution and all associated problems such as synchronizing distributed control flow is not addressed by their work. (ii) The architecture employed by the ReSpecT-based BPEL engine puts the activity logic into the tuplespace itself (through reactions), rather than decomposing it and moving it to tuplespace clients. Tuplespace clients do not implement transitions, but rather are external entities that simply consume and produce tuples. It is the tuplespace that executes reactions implementing BPEL activities which are triggered if certain tuples are present in the tuplespace. As a result, the ReSpecT engine is not decentralized: BPEL logic is executed centrally in the tuplespace in form of reactions. Furthermore, (iii) it is unclear how or if join variables (as discussed in Chapter 6), a fundamental feature required to reuse activity implementations across process instances, are supported. It is also not mentioned how *conflicts*, i.e. overlapping or even mutual exclusive requirements of different clients, are resolved. Both problems are explicitly addressed by the `sync` operation in our work. Finally, (iv) the similarity of the intermediary graph format to Petri nets is not exploited in their work. The format does not have formal semantics and seems to only play a very minor role in the architecture. EWFNs in contrast are first class citizens in our architecture, have formal semantics and can be used for verification and validation purposes through their defined mapping to "traditional" Petri nets. Moreover, EWFNs can serve as a generic specification language for tuplespace-based interactions, something that is also not addressed by the work on ReSpecT.

The second category of tuplespace systems is operation-based, which is also the category our tuplespace implementation, called the "Process Space", described in Chapter 7 belongs to. Specific to our implementation are multi-tuple operations, especially `sync`, a coordination primitive that allows for synchronized consumption of a set of tuples with the support for conflict detection and resolution. The need for operations involving queries on multiple tuples however has been addressed in tuplespace research before. In the following, we analyze operation-based tuplespace implementations that support multi-tuple operations and compare them to our implementation.

Jada [CR97] In Jada, both disjunctive and conjunctive queries on tuples

are supported by the `getAny` and `getAll` operations respectively. The disjunctive query operation allows a user to define a set of templates and returns all tuples matching *any* of the defined templates; the conjunctive query operation returns only those tuples matching *all* of the defined templates.

Although the `getAll` operation – by allowing a user to define a set of templates that have to be fulfilled at the same time – is somewhat similar to the `sync` operation proposed in Chapter 6, it is different with respect to the result of the matching operation. While the result of the `getAll` operation is a set of tuples in which *each individual tuple* matches *all* of the defined templates, the result of the `sync` operation is a set of tuples, where there is *exactly one* match for *each of the defined templates*. Since the latter semantics is an essential requirement for e.g. realizing synchronizing join operations, Jada's `getAll` operation is not suitable for that purpose.

Workspaces [Tol02, TG01] Workspaces, a workflow engine based on coordinated XSL transformations of XML documents built on top of a tuplespace infrastructure, recognizes the need for splitting and merging or joining control flow by parallel consumption of one document for each execution path to be joined. While this might suggest the use of a modeling language that follows the tuplespace paradigm of consuming and producing tuples, the authors instead choose to implement a proprietary process definition language without precisely defined semantics.

The concept behind the join operation in workspaces is conceptually similar to the `sync` operation, Workspaces however lacks a detailed description of its semantics and does not elaborate on its realization, e.g. how conflicts are detected and resolved. In addition, it does not cover the case of join operations spanning multiple distributed tuplespaces. Generally, Workspaces is not a distributed workflow system, it is based on one central tuplespace that coordinates all executed process instances.

RLinda [FÁBE06] and DRLinda [FÁE07] Renew-based Linda, or RLinda for

short, is a tuplespace system that is implemented using executable Petri nets called reference nets [Val98], a subclass of the nets-within-nets paradigm. Reference nets can be directly executed using Renew [KWD+04] and are used to model the semantics of the tuplespace and of its coordination primitives. As a result, the tuplespace itself is modeled in executable reference nets and executed using the Renew Petri net interpreter. DRLinda (distributed RLinda) builds upon RLinda and introduces runtime configurability and distribution, again, implemented using reference nets and executed using Renew. For distribution, clients connect to a single DRLinda coordinator that acts as a load balancer and forwards client requests to corresponding RLinda tuplespaces.

The most substantial difference of RLinda and DRLinda compared to our work is that Petri nets are used to implement the tuplespace rather than to model the interactions between the tuplespace and the clients it coordinates. Although the authors claim that both, RLinda and DRLinda are suitable for business process execution, they lack a proof how non-trivial workflows (e.g. supporting long-running transactions, error handling, correlation, etc.) that go beyond using simple control flow directives can be implemented using their systems.

Bonita [RW97] Although Bonita does not address the problem of synchronized consumption of tuples, it enables – through its operations `dispatch`, `arrived` and `obtain` – a more efficient implementation of this task than the original Linda dialect does. By issuing a call to `dispatch`, a user can express its interest in a certain template to be matched on a particular tuplespace. The return value of the `dispatch` operation is a handle that can be used in `arrived` to check whether a corresponding tuple is available and by `obtain` to actually retrieve the tuple. Separating the "check for existence" from the actual retrieval operation allows for shorter transactions when a synchronized join is implemented using Linda primitives as described in Algorithm 6.15. However, this approach still involves transactions being unnecessarily rolled back to resolve conflicting requirements. This is an inherent problem of this

approach – no global knowledge is available on the client-side.

TSpaces [LMW99] and the TSpaces Services Suite[FLN+03] TSpaces is a tuplespace implementation by IBM Research. It was created from the idea to merge basic functionalities from tuplespaces and databases and thus supports transactions, persistence and recovery. TSpaces also offer an extended Linda coordination model with multiset operations and – of most interest for this analysis – support for the query type `andQuery`.

While at first sight this looks very similar to the semantics of the `sync` operation, it is in fact quite different: an `andQuery` *combines* the results of individual queries, e.g. only tuples that match *all* given query expressions are returned. The purpose of `sync` however is not to combine queries but to *synchronize* the arrival of multiple tuples, e.g. to wait until a specified set of tuples is available in the tuplespace. The templates used to identify the set of tuples to wait for may even be disjoint.

The "TSpaces Services Suite" (TSSuite) provides a Web service infrastructure based on TSpaces tuplespaces, including an implementation of UDDI [B+02]. TSpaces is also used as the transport mechanism for Web service communication using TSSAPI, the TSpaces Service API[1]. Additionally, the TSSuite offers a set of tools and libraries to support service deployment, UDDI-based discovery and service invocation.

Although the TSSuite explicitly suggests the use of multiple tuplespaces distributed over the network, there is no support for synchronized access to different tuplespaces, not even using a transaction (distributed transactions are not supported). Web service compositions are hard-coded client applications, there is no support for a higher-level Web service composition language such as WS-BPEL. As such, the TSSuite can be seen as a first step towards Web service integration of tuplespaces, but does not go as far as supporting compositions of Web services in the form of business processes nor their distributed enactment.

XVSM [KRJ05, KMS08] The concept of Extensible Virtual Shared Memory

[1] Usually, this would be implemented as a separate Web service binding.

(XVSM) describes a general purpose coordination middleware and application protocol that is designed as a distributed virtual shared memory system. A reference implementation of the XVSM model is available by the name of MozartSpaces[1]. In this analysis, we focus on the XVSM concepts of *Container*, *Coordinator* and *Selector*, as these are the parts where functionality required to implement synchronization of control flow could be implemented.

A Container in XVSM provides a (optionally bound) scope for tuples, similar to a tuplespace; it allows to define one or more Coordinators, which influence the semantics of tuple access operations on the Container, e.g. tuples accessed through the FIFO Coordinator are accessed in the order they were added (first-in-first-out) to the Container. Predefined Coordinator types include Random, FIFO, PRIO, Vector and Linda. In essence, Coordinators are an abstraction of the coordination model used and thus provide a flexible means to extend XVSM functionality by implementing new coordination models. Selectors are the counterpart of a Coordinator, i.e. they provide the necessary information for a Coordinator to select tuples based on the information contained in the Selector. When accessing tuples using a Linda Coordinator for instance, the Selector contains the template used to find matching tuples. Selectors can also define how many matching tuples should be retrieved using the `count` property. Furthermore, Selectors can be chained (similar to sequentially applying a chain of filters) to express complex queries [KMKS09].

Since XVSM is a generic middleware, it emphasizes on providing abstractions and extensibility. Only basic coordination models such as FIFO, PRIO or Linda are shipped with the reference implementation. Higher-level functionality such as operations for the synchronized consumption of tuples are not part of the MozartSpaces implementation. Due to its extensibility however, higher-level functions such as `sync` could be implemented as a custom extension in form of a SYNC-Coordinator plug-in

[1] `http://www.mozartspaces.org/`

and a matching Selector implementation.

A generic method for mapping workflow descriptions to Linda coordination primitives is described in [D+05b], which also presents an algorithm for mapping process models in form of UML activity diagrams to events communicated via a shared memory space called “object space”. Essentially, this shared memory space supports basic tuplespace operations. The authors however do not consider the use of Petri nets for the definition of their processes and do not examine the close relationship between tuplespaces and Petri nets. Moreover, this work clearly lacks the aspect of decentralization; the object space is a central “object database” used to coordinate execution steps defined by activity diagrams.

The subject of modeling tuplespace-based applications has been addressed in [AR01] where UML-SPACES, an UML profile that facilitates state-machine-based modeling of tuplespace applications is introduced. While their approach integrates well with existing UML tooling, it tries to fit the tuplespace coordination paradigm into UML Class diagrams, making it impossible to model the tuple flow between individual components – one of the main advantages of our Petri net based approach. Moreover, the semantics of our diagrams are formally defined and can be mapped to traditional PT-nets, allowing for reuse of existing Petri net analysis tools and algorithms.

In follow-on work [RA01], the authors introduce a new UML diagram type called “behavior diagrams” as a complementary diagram type to UML statecharts. Behavior diagrams are used to model “active objects” and are suggested as a general modeling language for agents or mobile processes. Behavior diagrams differ from UML statecharts by introducing a new inscription language and by enforcing locality of the attributes of a transition. Although being a different kind of diagram, the differences to our approach discussed with their previous work still holds: behavior diagrams neither are formally defined and thus no means for automatic validation are possible, nor are they specialized enough to capture all aspects of a tuplespace-based program but are rather general purpose diagrams to model agent interactions.

2.2.1.2 Message-Oriented Middleware

Message-oriented middleware is considered to be the most similar kind of middleware compared to tuplespaces. This insight goes as far back as to the initial paper about the Linda coordination language [Gel85]. In Chapter 3 we further investigate the similarities of both technologies and provide a detailed comparison of messaging and tuplespaces based on EAI patterns. The comparison based on concrete patterns reveals differences not found in related work before.

In [AvdADtH05], the dimensions of decoupling *time*, *space* and *synchronization*[1] are formally defined using Colored Petri Nets. Additionally, existing communication middleware technologies and products such as MPI, RPC, CORBA, JMS, Websphere MQ and JavaSpaces were analyzed according to the decoupling dimensions defined before. Based on these, both MOM (represented by Websphere MQ and JMS) and tuplespaces (represented by JavaSpaces) are found to be identical.

[Ley06] uses a similar approach to compare MOM and tuplespaces: here, the comparison is based on the dimensions of *time*, *location*, and *reference*. Furthermore, advanced features that are not originally seen as an aspect of MOM are considered, e.g. the `peek` operation or "persistent connections". While it is found that tuplespaces support a way of interaction not supported by MOM directly (called the "multiple read" pattern), it is concluded that tuplespaces and MOM are essentially equal.

In [FKLT07], this work is extended by identifying further patterns that are *directly* supported by tuplespaces but require separate implementation efforts on top of MOM. Furthermore, a description of how tuplespaces and MOM can emulate each other's behavior is presented. Again, tuplespaces and MOM turn out to be very similar.

[1] The execution of the sending and receiving application does not have to be blocked during the act of message sending and message reception; e.g. after sending a message, control is returned back to the sending application immediately without waiting for the network stack to acknowledge for instance that the message was accepted for delivery.

2.2.1.3 Process Algebra

In this section, we discuss related approaches to our work on the decentralized workflow execution model called EWFN presented in Chapter 4 and on its tuplespace-based execution engine called "Process Space" (Chapter 6). EWFNs are based on Petri nets, a widely used and accepted formalism to model concurrent systems. Petri nets however are a subclass of a field in research known as process algebras, with prominent members such as CCS [Mil82], CSP [Hoa78] and the π-calculus [Mil99]. For the history of process algebras and an overview of different members of this area of research, the reader is directed to [Bae05]. [BGZ98] gives background on the relation of the Linda coordination language to process algebras. A general overview of coordination models is documented in [BCGZ01]. Specifically interesting is the proof on the turing-equivalence of the Linda coordination primitives, given ordered semantics [BGZ97].

The Join calculus [FG96] and its various implementations in traditional programming languages such as Join-Java [IK02] and polyphonic $C^{\#}$ [BCF04] is related to our way of modeling control flow synchronization in EWFNs and the execution using tuplespaces. In the spirit of `sync`, the Join calculus decouples transmission from synchronization so that synchronization can be done locally in the form of a conjunction of events, e.g. to only proceed if a message arrived on all channels. Synchronizing channels however is only one synchronization pattern that can be expressed using the Join calculus; it is a generic means to express higher level synchronization primitives such as monitors, shared buffers, etc. Therefore, its area of application is similar to what we aim to achieve with EWFNs and their tuplespace-based execution, specifically the `sync` operation. The Join calculus is a means to formally model concurrent systems with focus on the expression of control flow joins, like EWFNs in our case. It has a matching implementation in the form of extensions to traditional programming languages. The most significant difference however is its general focus, it is neither geared towards workflow or decentralized workflow execution, nor is there support for distributed joins as they occur in a decentralized setting, e.g. by joining paths of control flow implemented

by tuples residing in tuplespaces that are running on different nodes in the network.

Multi Agent Protocols (MAP), a π-calculus based language to define web service choreographies has been proposed in [BWR08]. MAP was primarily designed for the scientific field where large amounts of data are involved in processes and thus centralized enactment is not feasible. Instead, data is directly passed between "peers", avoiding a central hub where all data needs to be passed through. Messages between peers are exchanged in a non-blocking, reliable and buffered manner. The MAP language is used for both, modeling of the functionality of each peer and for execution. For model checking, MAP is translated to PROMELA [Hol91] and then fed into the SPIN [Hol97] model checker. An implementation of MAP is available as the MagentA Choreography Framework[1].

The most significant difference of the MAP approach compared to our work is that each peer is already modeled as a separate entity, i.e. when modeling the system, the user already decides how each partition of the process exactly looks like. This method of partitioning is similar to the "manual partitioning" approach in our work. However, there is no support for semi-automatic (with the option for manual corrections) or fully automatic partitioning as in our system. Furthermore, our system uses a standardized workflow language (WS-BPEL) as the language to model workflows and thus supports the full range of related tools such as visual editors developed for this language. This stays in direct contrast to the MAP approach, were a process is modeled directly in the MAP language itself.

Another related approach to EWFNs is the execution engine around Reo. Reo is described as "a paradigm for composition of software components based on the notion of mobile channels" [Arb04]. In Reo, higher level coordinators – called *connectors* – are built from simpler ones. *Channels* are the simplest connectors possible. Examples for higher-level coordinators are join, split, filter, FIFO, or ShiftFIFO. Components and channels are assumed to be mobile, i.e. if a component moves to another node in the network, the channel moves with

[1] `http://homepages.inf.ed.ac.uk/cdw/index.html#magenta`

it, maintaining the original structure of the network of connectors. Channels itself may be moved at runtime, e.g. a channel may be disconnected from one component and connected to another component, effectively changing the topology of the channel network. Operations possible on channels are `wait`, `take`, `read` and `write`, thus channels are exposing a coordination interface roughly similar to tuplespaces. [LA07] builds upon the work on Reo and uses it for coordinating the execution of Web services. Coordinations are expressed in WS-BPEL and then translated to Reo, model checked and executed in form of Java classes.

The most significant difference to our approach is Reo's focus. It is designed to be a generic coordination model, and was not specifically designed for decentralized workflow enactment. Therefore, the BPEL to Reo converter and the resulting Java classes execute the given BPEL orchestration in a centralized manner. Strategies and algorithms for process partitioning are not provided. Furthermore, the BPEL to Reo conversion only addresses basic control flow constructs, the implementation of advanced BPEL features such as compensation or DPE are not part of the translation.

2.2.2 Petri Nets in Workflows

Petri nets are an established formalism in the field of workflow management and are thus largely used for different purposes ranging from providing a model for execution over support for distributing a given process model to documentation of patterns and verification of process models. In this section, we analyze the state of the art of workflow research in the aforementioned categories and relate each of the approaches to the parts described in this work. In particular, we focus on the contributions of the Petri net based decentralized workflow execution model described in Chapter 4 and the BPEL to EWFN transformation in Chapter 5.

2.2.2.1 Petri Nets in Workflow Execution

The general application of Petri nets to workflow technology is studied in [vdA98]. Here, the authors argue that Petri nets are a suitable process mod-

eling language which – through its underlying formal model – allows for sophisticated process analysis techniques. This work also covers the basic principles how Petri nets can represent workflows.

Yet Another Workflow Language (YAWL) [vdAtH05] is a workflow language that took Petri nets as a starting point and added direct support for workflow control flow patterns [RtHvdAM06]. As a result, the notation of YAWL does not actually resemble a Petri net, but is rather a custom notation inspired by the various workflow patterns underlying its design. The semantics of each element is described in terms of a transition system, allowing for automated analysis and verification of YAWL process models. In our system, BPEL is used to define the process model and thus essentially plays the same role that YAWL does when modeling a process. Similar to EWFNs, YAWL also supports removing tokens from places, specifically to implement cancellation patterns [vdAADtH04]. The YAWL language comes with an execution engine[1] to enact XML representations of YAWL process models. The engine consists of several components such as a graphical designer, a process repository, worklist handler, Web service broker, etc. that communicate with each other using Web services. Users and applications taking part in a process are abstracted and implemented as Web services. Also, a transformation from BPEL to YAWL is available [BP06].

The most significant difference to our work is decentralization: neither the engine nor the BPEL transformation or the YAWL language itself is designed for decentralized execution. It is interesting to see however that some problems, like the implementation of cancellation in YAWL, is implemented in the same way in EWFNs through the introduction of functionality to explicitly remove tokens from places (cf. the *Stop Process* and *Stop Scope* EWFN places).

Reference Net Workshop (Renew) [KWD+04] is a modeling and simulation engine for the reference net [Kum02] (a.k.a. nets-within-nets) class of Petri nets where tokens may again represent Petri nets. In the form of a plug-in, Renew has been extended to implement a workflow engine [JKMUN02]. Compared to our approach however this engine lacks the aspect of decentralization and directly uses (high-level) Petri nets as the modeling language.

[1] `http://www.yawl-system.com/`

No transformation from a higher-level workflow language to reference nets has been defined. [ÁBE05] is another approach that implements Web service compositions on the basis of the nets-within-nets paradigm using reference nets. Here, Linda tuplespaces are used for the communication between the Web services involved. The tuplespace however only serves as an (asynchronous) communication buffer between the composed Web services. Neither the close relationship between Petri nets and tuplespaces is exploited, nor does the system provide support for a higher-level workflow language such as WS-BPEL. Furthermore, processes cannot be partitioned arbitrarily as in our system, peers must rather be programmed with concrete partitions in mind.

Another workflow execution engine based on Petri nets is pnengine[1] [DLO+08] which supports orchestration of RESTful services and uses *Service Nets* – a Petri net dialect supporting link passing mobility and value passing – as the modeling language. Process models are represented using a PNML-based[2] format, which is also used for execution. Similar to EWFNs, Service Nets recognize the need for read access to places, i.e. non-destructive access to tokens, to implement variable access for instance. While our token format is tuples, Service Nets directly use XML. Conflicts in tuple consumption are resolved by randomly rolling back database transactions. While pnengine directly executes Petri nets in PNML form, it was not designed to leverage the token passing technique of Petri nets to implement decentralized execution of processes. The engine itself executes process definitions in a centralized manner and is not able to hand over execution of a process instance to other engines in the network.

FlowManager [ACGV02] and XRL/Flower [VHvdA02] are both workflow systems where – similar to our approach of transforming BPEL to EWFNs – the semantics of the employed, higher level process language is expressed using Petri nets. XRL/Flower for instance uses the Exchangeable Routing Language (XRL) for process definition, which is transformed to Workflow Nets [vdA98] serialized as PNML and fed into the Woflan [VBvdA01] analyzer that checks the resulting Petri nets for consistency, deadlocks, lifelocks, etc. Finally,

[1] `http://code.google.com/p/pnengine/`
[2] `http://www.pnml.org/`

the PNML is executed directly by an execution engine that is based on the Petri net kernel (PNK) [KW01], a set of tools and algorithms that ease the development of Petri net based applications. PNK comes with modules and programming abstractions for visualizing and editing PNML, PNML parsing, and PNML simulation.

While both systems use a high-level language for representing process models which are then transformed to Petri nets for execution, they do not cover the main goal of our work: decentralized workflow execution. Both are centralized systems that do not exploit the similarity between the token passing semantics of Petri nets and tuplespaces.

2.2.2.2 Petri Nets in Workflow Distribution

[CFN07] introduces PN-Engine, a distributed WfMS that executes process definitions specified in Petri nets. The engine itself is based on Jini[1] as the service broker infrastructure and JavaSpaces [FAH99] as the communication middleware. The architecture comprises of a transition service for all transitions of the Petri net that notifies clients (called transition listeners) when a transition fires. Transition listeners may run on different machines and download the logic to be executed from a central repository using Jini facilities. The JavaSpace is used to store internal state and the marking of the Petri net, which is encoded according to model 2 described in Chapter 6. Follow-on work [CFN08] of the group around PN-Engine focuses on dynamic reconfiguration of workflows.

Comparing PN-Engine to our approach reveals several differences: (i) most notably, the engine is not decentralized as in our sense, arbitrary partitioning of the process is not possible. In fact, the process model itself cannot be partitioned at all. Although the authors call the engine "distributed", the component that executes the Petri net is central, i.e. the net is always stored in a single JavaSpace. The transition service and associated listeners may run on different nodes, just like the services orchestrated by BPEL may each reside on a different remote location. (ii) Petri nets are used for modeling *and* execution, no higher-level workflow language (e.g. BPEL as in our work) is supported.

[1] `http://www.jini.org/`

(iii) No solution to the `sync` problem to implement joins or even distributed joins (cf. Chapter 6) is presented.

In [GLO98], a concept for a decentralized workflow engine based on partitioning of Petri nets is introduced. The resulting system is described as an evolution of distributed database management systems. Data partitioning (i.e. horizontal and vertical partitioning in distributed databases) and replication concepts are extended to process partitioning and replication. A high-level Petri net dialect called Predicate/Transition Nets [Gen86] is introduced as a natural extension of the relational data model to formalize dynamic behavior of relations. Places represent relations, a transition represents an operation on the relations in adjacent places and tokens represent tuples. Techniques for horizontal, vertical and diagonal partitioning of workflow graphs are introduced. These techniques are designed to fulfill necessary requirements like completeness, minimality and disjointness.

Despite being based on distributed databases rather than using tuplespaces, their concepts have some similarities with our approach. Just as EWFNs are a Petri net dialect that was designed to model tuplespace-based interactions, they use Predicate/Transition Nets that incorporate the capabilities (defined by the relational model) of the underlying execution middleware. However – as the authors write themselves – they have yet to tackle the problem of (distributed) joins. Moreover, no support for a higher-level workflow language such as BPEL and a corresponding translation to the native execution model is given.

2.2.2.3 Petri Net Transformations of WS-BPEL

There are currently two feature complete (including all BPEL aspects like fault and compensation handling, correlation, DPE, etc.) transformations from BPEL to Petri nets available [LVD09], one created by members of the universities Berlin and Rostock (abbreviated by HR) [HSS05], the other one by members of the Queensland University of Technology (QUT) [OVvdA+05a].

The two most challenging problems when transforming BPEL to Petri nets are: (i) to find a suitable model to express Dead-Path-Elimination (DPE) [LR00] and (ii) to find a way to express global state changes, i.e. the transition from positive

control flow to fault handling where all running activities within a `scope` must be terminated immediately before execution can proceed. Interestingly, both HR and QUT present different solutions to this problem [LVD09]. While QUT relies on (scope-) global places to model both, DPE and termination (e.g. the places *to_stop* and *to_continue*), the HR approach uses colored tokens to model DPE and is terminating (employing the so called "stop pattern") all activities within the `scope` concurrently, using the places *stop*, *fault*, *fault_in* and *fault_up*.

In a sense, both approaches are workarounds for a generic problem exposed by these kind of mappings; there is a fundamental mismatch between languages like BPEL requiring global state to be modeled, and Petri nets that are designed around the notion of locality (i.e. local actions have local effects) and explicitly discourage the use of global state.

Further differences of the QUT and HR approaches are that QUT abstracts from data right-away, i.e. the transformation does not take variables, messages, correlation sets, etc. into account. HR in contrast provides a range of patterns for each BPEL activity, the most complex ones also include data. When it comes to analysis however, the patterns that abstract from data are chosen.

Moreover, the focus of both transformations is different: while QUT focuses on analyzing *soundness* of the workflow models, HR concentrate on the analysis of their communication behavior. As a result, QUT employs a Petri net dialect called Workflow nets (WFN), whereas HR are using open Workflow nets (oWFN) an extension of WFN that allows to model interface places needed to focus on the interaction of the modeled processes.

A Workflow net [vdA98] is an extension of classical Petri nets that introduces a distinct source and sink place (typically called i and o). *Soundness* is a behavioral property of Workflow nets denoting that all paths of the net lead to proper completion, i.e. all places except place o are empty (not marked) and there is no dead transition. The tool developed to check soundness on BPEL processes is called WofBPEL [OVvdA$^+$05b]. Besides soundness, WofBPEL is also able to detect unreachable and conflicting message consumption activities.

Open Workflow nets [MRS05] are an extension of Workflow nets that introduce *interface places* and a description of the set of desirable end markings.

There are several tools around oWFNs: BPEL2oWFN [LMSW06] transforms BPEL into oWFNs and checks 54 of the 94 static validation rules specified in the BPEL specification [A+07]. LoLa [Sch00] is a Petri net model checking tool that can validate properties such as deadlock, boundedness, liveness and reachability. Fiona [LMSW06] analyzes the communication behavior of BPEL processes, especially *controllability* and is able to synthesize a partner process from a given BPEL process. Furthermore, it can calculate an *operating guideline* that characterizes partners that communicate deadlock-free with the original process.

The key characteristic of the presented approaches mapping BPEL to Petri nets is that none of them is geared towards workflow *execution*; both clearly focus on verification, e.g. statically checking properties such as soundness, correctness and reachability. The resulting models thus are allowed to contain simplifications (and are in fact making heavy use of it) such as non-deterministic choice constructs and lack explicit modeling of data flow. Our approach in contrast focuses on being directly executable and explicitly disallows simplifications. Furthermore, our patterns not only cover the entire data and control flow, we also provide the structure of the tokens communicated. The EWFN patterns described in Chapter 5 document how BPEL activities can be implemented using the coordination primitives of a tuplespace. Our patterns can thus be seen as a blueprint for the implementation of tuplespace client applications that implement the functionality of BPEL activities. This is in contrast to the QUT and HR patterns, which are designed for computer-aided verification and thus try to abstract from implementation details. In order to reduce the number of elements in the Petri net and by that to reduce the state space to be explored by the model checker, they even abstract from data and solely focus on control flow. Decisions expressed by an `if` activity for instance are reduced to a non-deterministic choice, i.e. two arcs leaving a place. Furthermore, before actually enumerating the state space or calculating transition invariants, Petri net reduction techniques are employed to further reduce the size of the net to be analyzed.

Apart from the fundamental different intentions between the patterns, there are a number of techniques we could learn from their work. These include: (i)

the methodology to create composable patterns through overlapping of places, (ii) the way of implementing DPE by propagating tokens of different colors to indicate positive or negative control flow, and (iii) the method to implement termination, i.e. the "stop pattern". In our work, this is implemented as a separate pattern that is behind the *Stop Scope* and *Stop Process* places. The original idea to realize Dead-Path-Elimination by using "dead tokens" in Petri nets was introduced as *boolean networks* in the context of event-driven process chains (EPC) in [LSW97].

Another source of Petri net patterns for workflows are the so called *workflow patterns* [vdAtHKB03, RtHvdAM06]. They describe generic patterns with the goal to evaluate the expressiveness of different workflow languages. Additionally, the language YAWL [vdAtH05] arose from the work on these patterns to create a workflow language that implements most of them.

In this work, we use workflow control flow patterns to demonstrate the expressiveness of EWFNs, and also to define a test process that covers all basic patterns to evaluate our tuplespace kernel implementation.

2.2.3 Distributed Workflow Enactment

In this section, we give an overview on related distributed workflow management systems and analyze the differences to our work. We specifically concentrate on systems using BPEL as the execution model (Section 2.2.3.1), Agent-based systems (Section 2.2.3.2) and Grid/Peer-to-Peer based systems (Section 2.2.3.3).

2.2.3.1 Systems Using BPEL itself as Execution Model

In this section, we discuss a number of approaches using BPEL directly as the execution model, e.g. by splitting and rewriting (i.e. introducing or changing activities) the original process into multiple BPEL processes. Subsequent sections then focus on systems that use lower level execution models requiring a transformation from BPEL into these models.

In [NCS04], decentralized execution of BPEL processes is motivated by minimization of communication cost and maximization of throughput of multiple

concurrently running instances. An algorithm for automatic decomposition of BPEL programs into individual partitions is presented along with a cost model to aid the decomposition algorithm. The result is a set of individual BPEL processes that, when executed using a modified BPEL engine, implement the same semantics as the original process. The execution model therefore is BPEL itself as the original process gets analyzed and rewritten in form of multiple processes which connect to each other by `invoke/receive` pairs. [CCMN04] gives further details on the implementation of the engine, which is based on BPWS4J [CKNW06], and discusses build- and runtime issues of their implementation. Communication between each of the partitions is facilitated through asynchronous messaging, specifically a SOAP/JMS[1] binding.

It is very interesting to see that their main problems lie in the implementation of fault- and compensation handling. Extra fault-handlers and fault-forwarding activities need to be introduced to propagate faults within split scopes. Furthermore, a central "monitor" that polls the runtime databases of all included BPEL engines is required to maintain information about global state changes, as required e.g. by `scope` termination and to coordinate `scope` compensation.

The very specific problem of implementing Dead-Path-Elimination (DPE) in decentralized workflow management systems is discussed in [Yil07]. Additionally, the author generalizes the concept of DPE to cover both control- and data flow, i.e. to deactivate activities that depend on data generated by others that were skipped.

In [BMM06], a similar approach to distribute service orchestrations in BPEL form based on partitioning rules is discussed. The presented partition rules operate on a BPEL process, producing a number of BPEL processes that together implement the semantics of the original process. Again, `invoke/receive` and `invoke/pick` activity pairs are used to connect the individual process partitions. In this work, many interesting (and hard) points are unclear or not mentioned: it is neither explained how split scopes or loops are synchronized nor how or if Dead-Path-Elimination is implemented.

The third approach in this discussion for decentralized BPEL execution using

[1] `http://www.w3.org/TR/soapjms/`

BPEL itself as the execution model is [KL06] which is further detailed in [Kha08]. Here, *a user* splits a BPEL process model into a number of partitions which then result in a corresponding BPEL process each, along with necessary deployment information. Exchange of process instance data and passing control flow between partitions is realized by introducing additional activities in each partition (called "receiving flows", since these are also capable to correctly forward DPE between partitions), similar to the `invoke/receive` pairs in the approaches discussed before. This work poses a very interesting question: how far can you go in splitting BPEL processes without requiring extensions to the execution engine itself, i.e. by using only BPEL constructs to implement the semantics of a central BPEL process? It turns out that it is the semantics of `scope` (specifically fault and compensation handling) and loop (e.g. `while`, `forEach`) activities that cannot be emulated with pure BPEL. Here, a central coordinator is required that provides the means to handle such cases. As a result, the answer to the initial question is that you need extensions as soon as one of the offending constructs is used in the process model to be split.

Another important issue solved by this work is the addition of explicit data flow links to BPEL, a fundamental requirement for partitioning of a process model. Data links are implemented in the form of BPEL-D [Kha07], a BPEL "dialect" that adds explicit data flow arcs. The idea is that, before the actual partitioning can take place, a given process is analyzed to extract data flow arcs from the variable accesses of each activity [KKL08].

The role of BPEL-D in this work is roughly equivalent to EWFNs: in both cases, data and control flow are defined explicitly and thus provide the algorithms that derive the partitioned processes with necessary information. Data flow in particular is a very important aspect in distributed systems as it provides many important hints for optimizations, e.g. minimizing the size of the data shipped between activities.

To summarize, there are three main problems when using BPEL itself as the execution model for decentralized execution of BPEL processes:

Complex semantics of split scopes and loops The analysis of all approaches using BPEL as execution language has shown that it is especially hard to

maintain centralized semantics of split `scope` or `loop` activities. Here, it is necessary to monitor and forward state changes that are internal to activities and thus to the execution engine (e.g. the passing control flow within a split `loop` or passing on termination between members of a split `scope`). Clearly, these effects cannot be monitored on the level of BPEL. For this reason, all approaches either disallow splitting these activities or introduce a central entity (called "monitor", "scope registry" or "coordinator" in approach one, two or three respectively) that hooks into the execution runtime and distributes state changes to each member of a split `scope` or `loop`.

Lack of explicit data links A generic requirement for all approaches using BPEL is data flow analysis, since BPEL does not provide explicit data links. Instead, data is accessed through scoped variables. Decentralized execution engines therefore need to analyze a given BPEL process first (e.g. using the algorithms presented in [KKL08]), and then add the derived data links to the process model before the actual splitting can take place. A variant of BPEL, called BPEL-D [Kha07] has been proposed for this purpose.

Synchronization of variables in split scopes BPEL does not allow variables to be accessed remotely. If a `scope` was split into multiple partitions, each partition has a local copy of the variable. Complex mechanisms – again implemented in pure BPEL when using BPEL as the execution model – are required to keep all local copies of the variables in sync.

2.2.3.2 Agent-Based Systems

In [BVV03], the general vision of agent-based execution of BPEL workflows is introduced. The authors foresee BPEL as the language to specify the "initial social order" of agents. In this work, agents are proxies to Web services, the initial social order thus is the Web service orchestration logic defined by the process. The idea is that the workflow may evolve and adapt to changes in the environment through proactive agents. The paper also analyzes differences

between the interaction protocols defined by the Foundation for Intelligent Physical Agents (FIPA) and the BPEL language.

Based on these initial ideas, a corresponding agent-based BPEL execution engine is described in [BV04]. The system focuses on adaptability after deploying the initial agent configuration in the form of a BPEL process definition. Before execution, a BPEL process is transformed into Colored Petri Nets where transitions are implemented by agents and places specify pre-conditions for the agents to run. The execution itself is carried out using a central XML database. The engine is tightly integrated into a FIPA runtime, e.g. services such as the directory facilitator for agent naming and lookup and the message transport service are used.

[GRCB05a, GRCB05b] describe a distributed multi-agent workflow enactment service that interprets BPEL using the Lightweight Coordination Calculus (LCC). Agents are implemented as "active" Web service proxies, i.e. they execute parts of the orchestration logic upon reception of a message. Messages passed between agents contain the interpreter itself (in form of LCC expressions), the current execution state (including process variables) and the process definition (in BPEL form). They then execute parts of the orchestration logic, serialize everything into a message again and pass on execution to the next agent. That way, orchestration logic is "moved" to the Web services. Additionally, an algorithm to transform a BPEL `flow` into a `sequence` is provided, choosing one particular order of the different possible execution orders of parallel activities.

[JPPMMJ08] introduces ZenFlow a BPEL engine based on "reflection". Reflection is an architectural property that allows a system to be observed and modified during runtime through its so-called *meta interface*. Or, put more abstract: "Reflection refers to the ability of a system to reason about and act upon itself" [JPPMMJ08]. The tasks a reflective system can fulfill are similar to what can be done using Aspect oriented Programming (AOP). In ZenFlow, metaobjects allow to introduce additional functionality before or after a BPEL activity was executed. This ability, together with Java RMI is used for distributed process execution: the metaobject acts as a load-balancer and forwards process execution requests to other ZenFlow engines. The execution

state is moved to a “worker BPEL engine” that runs the requested process and moves the final state back to the metaobject, which then gives the answer to the client.

Interestingly, all of the discussed systems use a different execution model: either a coordination calculus (LCC), Colored Petri Nets or metaobjects. None of the approaches however gives details on how the transformation from BPEL into the low-level execution model works and how specific problems identified in the previous section such as the implementation of DPE or how functionality required to coordinate split scopes and loops is implemented. Additionally, two of the three systems are not decentralized in our sense: [BV04] uses a central XML database to exchange information between agents, [JPPMMJ08] uses load-balancing and thus distributes the execution of process instances to different nodes, but does not allow to execute parts of the orchestration logic on different nodes. In the case of the system that used Colored Petri Nets as execution model, no information is given about the transformation problems specific to BPEL to Petri net transformations like the mismatch of global BPEL semantics and the local semantics of Petri nets.

2.2.3.3 Grid/Peer-to-Peer Workflow Systems

Grid or Peer-to-Peer workflow systems are decentralized by definition: their use case is very similar to the third scenario presented in Section 1.3. One of the biggest challenges the Grid faces is to avoid unnecessary remote communication, especially with the large amounts of data involved [FK04] in typical scenarios. A recent development in this area is the use of workflow technology in scientific computing, e.g. to support large simulations or data analysis tasks.

In [Yu07b, Yu07a] a peer-to-peer BPEL process engine is introduced. The execution engine is based on *continuations*, a technique to represent the current processing state (e.g. program counter, call-stack, heap, . . .) in a serializable manner. Continuations provide programs with the ability to stop and save execution at any point and to return to that point later in time. The general idea thus is to serialize a running process instance, ship it to another node and continue execution from there.

The execution model in this work is based on an abstract state machine called CEKK, its atomic elements are states and transitions. State transition rules implement the semantics of BPEL activities; a `flow` activity for instance initiates as many parallel branches in the state machine as there are activities without incoming links. The corresponding runtime environment is implemented in the form of "Process Containers" that use message queues for incoming messages. A message itself contains the continuations, i.e. all necessary information to continue execution of a process instance. There are several interesting points in this work: (i) a process container offers two special actions, `stopall` and `compensateall` to implement the semantics of a BPEL `scope`. `stopall` stops all running activities in a `scope`, `compensateall` triggers compensation on all active child-scopes of a particular `scope`. BPEL scopes are managed by so called "scope agents", system-central components that monitor `scope` states in order to be able to implement `stopall` and `compensateall`.

In [Sch04], OSIRIS, a distributed workflow engine based on the concept of a *hyper-database* is presented. The hyper-database is a distributed peer-to-peer database system that offers a number of global services with ACID transactional semantics. Each node participating in the process execution runs a hyper-database layer that connects to the network of other nodes taking part in the process execution. A distributed publish-subscribe infrastructure is used for keeping process instance data in sync on all participating nodes. In OSIRIS, process instance data is migrated between nodes during execution of process instances. The data structure that contains process instance data is called the *whiteboard*. In case of multiple concurrent execution paths, the whiteboard is split and distributed amongst all parallel execution partners. The approach of distribution thus differs significantly from our approach, since instance data is migrated between peers, requiring complex and expensive mechanisms to merge data again after splitting for parallel execution.

The employed execution model however is very interesting. It is based on the notion of *flexible transactions*; the central property of a process is *guaranteed termination*. The combination of compensateable and repeatable activities in the process definition guarantee that a path to a final process state can be reached from every state in the process. This property can be statically

analyzed during design time. The call semantics for an activity are request-response: either an error message or a return value is returned. Additionally, each activity invocation fulfills ACID properties. Activities themselves are classified into: (i) *Repeatable activities*: an error in this kind of activity can be compensated by repeated execution. After a finite number of repetitions, successful execution is guaranteed. (ii) *Compensateable activities*: the effects of an activity can be compensated by executing a corresponding compensation activity. Compensation activities itself are required to be *repeatable*. (iii) *Pivot activities*: an activity is pivot if it is not compensateable. On execution errors, alternative paths in the process model must ensure reachability of a final activity.

In [BD00, BD97], $\text{ADEPT}_{\text{distribution}}$, a distributed variant of the ADEPT WfMS is introduced. Similar to other Grid WfMS, its primary goal is to minimize the amount of data shipped between systems involved in the orchestration. In this system, static (i.e. defined by the user) or variable (i.e. based on probability distributions of activity executions) assignments of tasks to WfMS servers are supported. The ADEPT execution model is designed with the support for dynamic, ad-hoc changes in mind – the original goal of the ADEPT WfMS. It is a formal model that is based on *symmetrical control structures*. Task sequences, branches and loops are *symmetrical blocks* with well defined start and end nodes. A workflow model is a directed, structured graph with nodes as tasks and edges as the control dependencies between them. Failure edges allow to model alternative paths to handle faults. Data flow is modeled by means of global variables.

Scientific workflow management systems are specialized WfMS for deployment and execution of large scientific experiments in Grid environments. Naturally, one of their primary goals is to avoid unnecessary shipping of data between nodes in the Grid. The most significant difference to “business” WfMS is their inherent orientation on data flow instead of control flow. Scientific processes usually are “data oriented”, instead of “control flow oriented”. This is due to the fact that data is the central entity in the workflow models. A process typically is comprised of a list of functions applied to the data, such as filtering, analysis, compression, visualization, etc. The programming model

thus typically consists of data sources and sinks, individual functions applied to the data and "data pipelines" connecting the functions.

Triana[1] [C+06] is a "Graphical Problem Solving Environment", that supports scientists in the composition of applications. It provides a graphical editor and an execution server that allows users to model workflows in form of directed, acyclic graphs which are serialized to a proprietary XML format. Individual applications (called tasks) are combined using pipes that offer either synchronous or asynchronous access to the buffered data. Tasks can be grouped together and assigned to certain nodes of a cluster; the groups can then be executed in parallel if no data dependencies exist, or in a pipelined fashion. The Triana system also provides various services by interfacing with the underlying Grid middleware, i.e. support for dynamic allocation of resources, clustering, monitoring, etc.

Kepler[2] [L+06] emphasizes on the use of channels as the means to connect individual tasks in a scientific workflow. Two fundamentally different workflow execution models (called "Directors") are supported: *Process Networks* send tokens through unidirectional channels acting as communication buffers. Writing to a channel is asynchronous (i.e. non-blocking); read operations in contrast may block, e.g. until data is available. *Synchronous Data Flow Networks* in contrast use fixed token production and consumption rates. Essentially, there is a central clock that defines the execution frequency of all involved tasks in the workflow.

Pegasus[3] [D+05a] specifically targets the problem of resource allocation and distribution of scientific workflows in Grids e.g. the task of finding a suitable mapping for workflows and their tasks to the physical resources available in the Grid in order to minimize execution time. It provides methods to restructure and cluster tasks in the original workflow, select appropriate resources and to control data replication in the Grid. For that, Pegasus closely interfaces with the underlying Grid middleware (e.g. provided by the Globus Toolkit [FK97]), and uses e.g. Grid job schedulers like DAGMan [Fre02] or Condor-G [FTL+02]

[1] `http://www.trianacode.org/`
[2] `http://kepler-project.org/`
[3] `http://pegasus.isi.edu/`

to execute the execution plans it calculated.

2.3 Conclusion

The goal of this chapter was two-fold: first, we introduced four main technical areas this thesis builds upon, namely message-oriented middleware, tuplespaces, Petri nets and Web services. We presented the Web service technology stack, highlighted important standards for this work such as WS-BPEL and provided background information on the main technologies used.

Second, we discussed related work in the three scientific areas the contributions of this work belong to: (i) in the area of coordination languages we emphasized on the novelty of the `sync` primitive, especially its abilities to efficiently resolve conflicts. Furthermore, we highlighted the insights provided by our work on the comparison of message-oriented middleware and tuplespaces based on EAI patterns. We also discussed the advantages of EWFNs as formalism and modeling language for tuplespace-based interactions. (ii) In the area of the use of Petri nets in workflows we discussed different approaches that use Petri nets as execution model and took a detailed look at existing approaches to transform BPEL to Petri nets. The difference to our BPEL Petri net transformation is that existing approaches focus on verification and are thus created with a completely different goal in mind. They make heavy use of reduction techniques and abstract from data in order to create small Petri nets to minimize the state space required to be explored by model checking tools. (iii) In the area of distributed workflow enactment, different areas such as agent or Grid workflow systems were explored with a focus on the execution models these systems use. One particular interesting finding was that, although many systems directly use BPEL as execution model, this imposes heavy restrictions on how individual partitions can be created. Certain activities such as scopes, loops or fault and compensation handlers can only be partitioned if central entities are introduced.

The analysis of related work has shown that our basic building blocks – Petri nets, tuplespaces and transformations from higher-level workflow languages into a low level execution language – are widely used in the architectures of

distributed workflow systems. Many of the problems we encountered were encountered by others as well, resulting in different proposals of corresponding solutions. These range from the introduction of multiset operations to tuplespaces, the implementation of DPE using colored tokens and token forwarding to address the mismatch of BPEL's global semantics versus the local semantics of Petri nets. This thesis builds upon these solutions and by that contributes to the state-of-the-art in the respective fields.

Chapter 3

Middleware in Workflow Systems

Workflow engines are very complex middleware systems that execute high-level programming languages and provide strong guarantees on quality of services. Their architectures heavily rely on underlying middleware such as relational databases, message-oriented middleware or application servers to implement persistence mechanisms, transactions, scheduling, etc.

In this chapter, we discuss the requirements on a decentralized workflow engine by starting with workflow systems in general and adding the requirements on the execution model of a decentralized execution engine. In Section 3.2 we review the architectures of existing workflow systems based on the discussed requirements and emphasize on their use of different kinds of middleware. We find that the navigator – the central component of a workflow system – is typically implemented using messaging middleware, but uses a particular design that does not allow decentralized execution. Based on the required decoupling dimensions for the decentralized components of our engine, we conclude that tuplespaces may be a viable alternative. Moreover, tuplespaces

provide a very natural way to implement the execution model based on token passing we foresee for our decentralized navigator implementation[1].

As a result, in Section 3.3 we compare messaging – the key middleware currently used for implementing the navigator component in workflow systems – to tuplespaces, the middleware we use for our navigator implementation. We start by a high level comparison of features which is refined by using EAI patterns, the primary domain of application for messaging systems, as basis for the comparison. We assume tuplespaces instead of messaging as the underpinning for the EAI patterns and in that way reveal differences not found in related work before. Finally, we compare tuplespaces with relational database management systems, a type of middleware that had substantial influence on the development of later tuplespace implementations.

Note that in this chapter, the terms "messaging" and "message-oriented middleware" are used interchangeably.

3.1 Requirements on a Decentralized Workflow Engine

In this section, we discuss operational requirements for workflow systems and their effects on the *execution model*, i.e. the way how a process is enacted.

3.1.1 Operational Requirements on Workflow Systems

[LR00] divides the requirements on a *production workflow system* (a workflow system that is integral part of an enterprise and is therefore mission critical to the enterprise itself) into two categories: *operational requirements* and *enterprise requirements*. Enterprise requirements concern security, administration standards and platforms and are not focus of this discussion. Here, we are interested in the operational requirements which will later be used to review the architecture of existing workflow management systems, especially to investigate how these requirements are implemented.

The following non-exhaustive list of operational requirements for a production workflow system is based on [LR00]. We summarize the requirements

[1] Details about our execution model and its mapping to tuplespaces are discussed in Chapter 4.

and briefly discuss how each of them are addressed by architectural properties, respective middleware such as message-oriented middleware (MOM) [BHR95] or relational database management systems (RDBMS) [SKS98], or both.

Reliability A workflow system needs to be able to gracefully recover from failure and start where execution left off. After failures, completed work must not be lost and partial work must be undone and restarted.

Typically, this is implemented by backing execution state with persistent storage and by using transactions for carrying out internal work of the workflow engine as logical groups of statements, e.g. atomically commit the results of a completed process activity. The application server(s) involved typically are stateless, i.e. they maintain all state information explicitly through resource managers. The employed transactions operate on the state managed by these resource managers and thus provide means for recovery and retry for the whole system.

Availability A workflow system must provide high availability. Sometimes, it might even be required that it provides continuous operation, i.e. to still provide service when the system is upgraded or parts of it are down for maintenance.

Availability is a property influenced by many different parts of the system such as hardware, system architecture and the software components used. However, MOM and RDBMS as underlying middleware systems typically support means for fail-over and fault-tolerance (e.g. through clustering and replication) and thus contribute mechanisms to improve overall availability of the system.

High Capacity A workflow system needs to support many concurrent users and process instances at the same time. Ideally, the system itself does not impose a limit on those resources.

Again, this property is highly influenced by the architecture of the workflow system and the capabilities of the individual software components used. However, using well established middleware with proven efficiency

and well-tested behavior under different load conditions helps building high capacity systems. Furthermore, MOM provides a programming model to handle concurrency and to avoid common problems such as throttling.

Scalability A scalable workflow system increases throughput (e.g. the number of requests satisfied) if additional resources are added. Scalability is a property of the structure of the workflow system, allowing to flexibly add additional resources and leveraging them upon availability.

Both, MOM and RDBMS typically provide means to balance work amongst parts of the system, e.g. through load-balancing/spraying components, through partitioning or both. Moreover, the MOM programming model inherently supports ways to balance the work amongst connected clients, e.g. through the *Competing Consumer* pattern [HWB03]. Also, the statelessness of the underlying application server(s) helps the load-balancing components since no server affinity is required for individual requests.

Traceability Traceability is a requirement of a workflow system to provide and persist information about the actions performed by the engine with regards to the processes executed, e.g. to monitor actions being performed on the level of the process. This can be later used for statistical analysis over completed process instances, for calculating costs related to the process, monitor KPIs and to check if business-level goals are met.

RDBMS are typically used for this purpose since they allow for persistent storage of structured data and to express complex, multidimensional queries e.g. through support for online analytical processing (OLAP).

Please note that many of the properties discussed are also influenced by the topology used for the different components the workflow system consists of. IBM for instance provides several topology suggestions[1] (commonly known as bronze, silver or gold topology) for the setup of WebSphere Process Server.

[1] See e.g. `http://www.ibm.com/developerworks/websphere/library/techarticles/0704_chilanti/0704_chilanti.html`

These include suggestions and best practices on how to create and configure clusters and partitions of the database systems, messaging middleware and application servers involved.

3.1.2 Requirements on the Execution Model

Clearly, it is much easier to implement a system meeting the requirements discussed in previous section in form of a *logically central*[1] system. The execution of business processes introduces a considerable amount of internal state that needs to be shared between the activities of a process. Furthermore, transactions to coordinate access to shared state can be implemented much more efficiently in a centralized system.

However, this stays in direct conflict with the primary requirement on the execution model of our proposed decentralized workflow system: *the avoidance of a central point of control*. There must not exist a single point in the system where all other parts depend on. Very often, such a central point exists in the form of a central database that is used to persist and exchange state, and thus is shared by all components of the workflow engine.

We solve this problem by employing *token-passing* as the underlying design principle of our execution model: a central point of control is avoided by transforming the workflow definition from a high-level language such as WS-BPEL into a set of fully decoupled, individual components that coordinate themselves by directly exchanging control and data-flow messages (tokens) – and in that way implement the semantics of the original process model. Each individual partition of the original process is called a *component* since it is implemented by pre-defined software modules configured in such a way that the semantics of the process partition they are from is re-created. Avoiding a central point of control also means that each of those components needs to be *fully decoupled* from all other components implementing the remaining partitions of the original process. Only in this case, each component can

[1] We define "logically central" to be the property of a system that forces clients to go through a central point during serving a request. Such a central point can be e.g. a load-balancer component in a database cluster or a "sprayer" component in clustered message queues. A fully decentralized system in contrast does not require such a component.

be deployed independently from all other components and thus the original requirement of being able to flexibly deploy a process model according to a given infrastructure without any restrictions imposed by the process model itself can be fulfilled.

However, employing token passing as the execution model also introduces certain drawbacks, including: (i) a higher complexity to debug and monitor the system due to the asynchronous and possibly distributed nature. Moreover, (ii) synchronous interactions require additional mechanisms to be implemented on top, e.g. *correlation* for synchronous request-reply processing. If shared state is required by the semantics of the construct to be executed (e.g. a shared variable or a shared context by activities) this has to be emulated using token passing. A possible solution can be using a shared token representing the state and update messages to represent state modifications.

In the following, we list requirements on the decoupling dimensions of our components that need to be met in order to be able to arbitrarily distribute them.

Reference Reference decoupling means that sender and receiver do not need to "know" each other in order to be able to communicate. Typically, this is achieved by sending and retrieving a message to and from a logical address, e.g. a queue name.

Time Time decoupled components do not have to be available at the same time in order to be able to communicate, i.e. communication is buffered. A message is delivered to its receiver when it comes online and is ready to receive messages.

Location In location decoupling, sending and receiving programs do not have to be in the same address space, machine or location. Communication is e.g. carried out over network connections.

According to [AvdADtH05, Ley06], both tuplespaces [Gel85] and message-oriented middleware [BHR95, HWB03] provide applications with decoupling in all dimensions discussed. We will come back to this observation in Section 3.3

where we take a closer look at these two particular middleware systems and their role in the architecture of a WfMS.

3.2 Review of Existing Workflow Engines

To validate our observations we review the architecture of prominent open-source BPEL workflow engines with focus on the use of middleware and how it is used to fulfill the requirements we have discussed before.

3.2.1 Stuttgarter Workflow Maschine (SWoM)

The "Stuttgarter Workflow Maschine"[1] or SWoM for short is a BPEL engine developed at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart, Germany. Its primary goal is to provide a BPEL engine that serves as a testbed for new research ideas and thus form the basis for different projects and diploma theses at the institute.

Figure 3.1 depicts the architecture of SWoM: it consist of three main building blocks, namely *Process Execution*, *Administration* and *Gateway*. Each of them communicates with the other building blocks using messaging middleware, indicated by the queue symbols between the navigator and the gateway or by the *manager topic* (depicted by the MT box), the publish-subscribe channel that connects the administration module with the module for process execution. SWoM makes extensive use of different middleware systems: it is based on an Application Server and uses a database system and messaging middleware for reliable inter-module communication and persistence. In the following, we summarize the use of each of those middleware systems in SWoM.

JEE Application Server

- The application server provides Web service endpoints for the process and its incoming message activities such as `receive`, `pick`, etc. It also offers functionality to act as Web service client, e.g. to implement outgoing message activities such as `invoke`.

[1] `http://www.iaas.uni-stuttgart.de/forschung/projects/swom/`

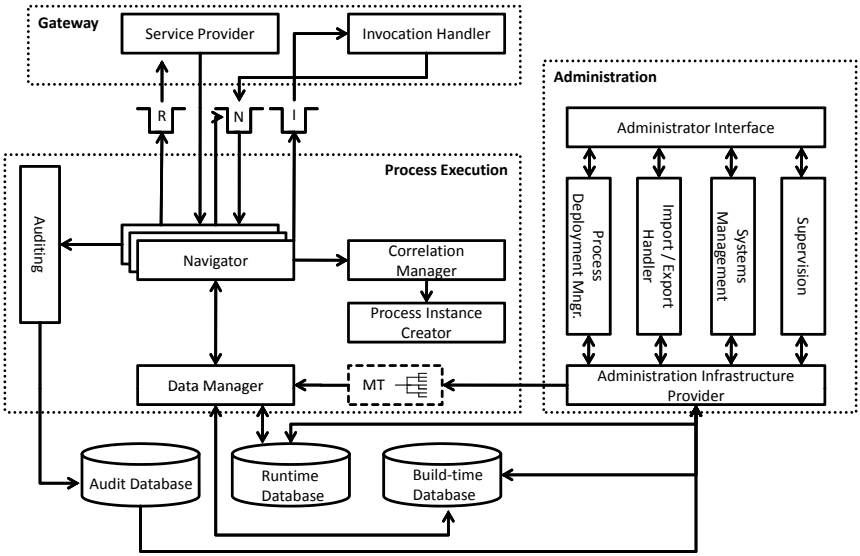


Figure 3.1: Architecture of SWoM (originally from SWoM 3, Design.pdf)

- The application server provides the functionality for the coordination of distributed transactions (i.e. a transaction coordinator) that are used between all middleware systems, e.g. the application server, the messaging middleware and the database system.
- It serves as a container for the application logic of the workflow engine that is implemented in form of Enterprise Java Beans (EJB). The service provider component that exposes processes as Web services for instance is implemented as a stateless session bean.
- The application server offers a JNDI implementation to manage datasources such as connections to the database or the messaging system
- The application server allows the workflow engine to schedule tasks through provided scheduling functionality. SWoM uses the scheduler for tasks that need to be suspended for a certain amount of time, to implement e.g. the BPEL `wait` activity.

Database Management System

- The RDBMS is used to persistently store (i) deployed process models in the so called *build-time database*, (ii) the state of running process instances in the *run-time database* and (iii) audit and logging information from running and completed process instances.
- It acts as a resource manager in distributed transactions that span all three middleware systems.
- The database management system provides query facilities for the analysis of information from the audit log database.
- It also serves as a persistent back-end for the navigator component.

Message-Oriented Middleware

- Instead of using a separate message-oriented middleware system, the messaging middleware of the application server is used.
- It provides the messaging functionality used for communication between each of the three building blocks of the engine. Persistent point-to-point channels (depicted as R, N and I: reply, navigator and invocation queue) are used between the navigator and the Web service gateway, persistent publish-subscribe channels (depicted as MT: the manager topic) are used between the administration and the process execution component.
- The MOM acts as a resource manager for the messaging channels included in distributed transactions.

3.2.2 Apache ODE

Apache Orchestration Director Engine[1] (Apache ODE) is an open-source BPEL engine hosted and developed under the umbrella of the Apache Software Foundation. A commercial variant is available under the name "Intalio|Server" from the company that originally contributed the ODE codebase to the Apache foundation.

[1] `http://ode.apache.org/`

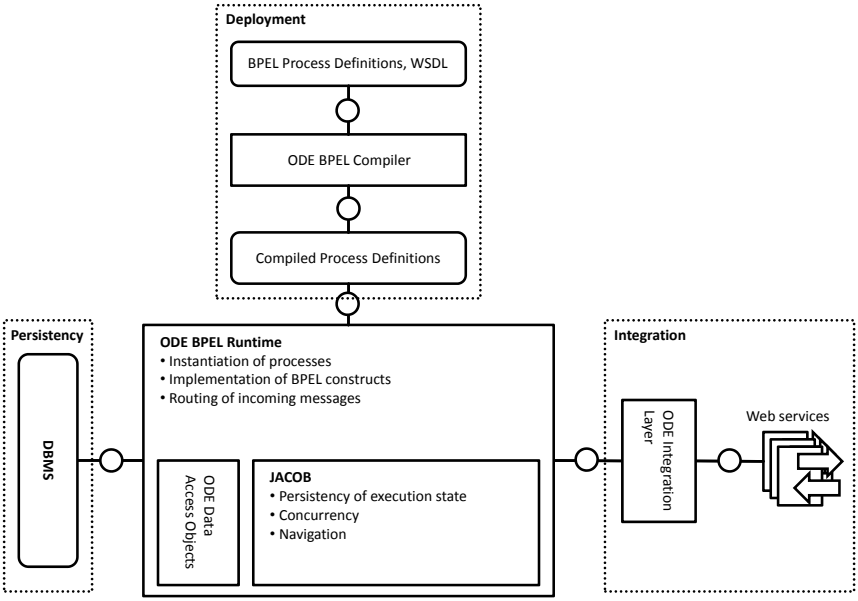


Figure 3.2: Architecture of Apache ODE (from `http://ode.apache.org/architectural-overview.html`)

Figure 3.2 depicts the architecture of Apache ODE. It is structured into four parts: (i) the process deployment module shown in the upper part of the figure, (ii) the ODE integration layer for interaction with the outside world on the right-hand side, (iii) the persistency layer depicted on the left-hand side and (iv) the ODE BPEL runtime in the center.

ODE does not specifically require an application server to run; all services it depends on, such as a Java Transaction API (JTA) implementation, are provided by the engine. For the communication with the outside world however, ODE relies on a so called *Execution Environment* that is abstracted by the ODE integration layer. The Execution Environments currently supported are either

JBI[1] compliant runtimes or Apache Axis2[2].

Execution Environment

- The Execution Environment provides communication channels for the process engine, such as handling incoming and outgoing Web service requests. Current implementations provide HTTP/SOAP connectivity directly through Apache Axis2 or indirectly through an JBI implementation that implements this binding.
- It schedules threads created by the ODE process engine.
- The Execution Environment manages the engine's lifecycle, e.g. it handles engine configuration and controls startup.

ODE Data Access Objects (DAO) abstract from a concrete database management system by providing an object model of the data to be persisted. Currently, there are two implementations of ODE DAOs: one for hibernate[3], a popular object-relational mapper and the other for OpenJPA[4], Apache's JPA compliant Java persistence solution.

Relational Database Management System

- ODE DAOs are used to transactionally persist: (i) the state of running process instances, including variables, partnerLinks, etc. and (ii) the state of the process execution engine itself, i.e. the state of the navigator component.
- Through DAOs, a RDBMS is also used as a persistent back-end for ODE's own implementation of a job scheduler.
- DAOs also abstract from distributed transactions using the Java Transaction API (JTA). Through JTA, the RDBMS acts as a resource manager for distributed transactions with the ODE execution engine (JACOB).
- Interestingly, deployed process models are not persisted using DAOs;

[1] `http://java.sun.com/developer/earlyAccess/jbi/`
[2] `http://ws.apache.org/axis2/`
[3] `https://www.hibernate.org/`
[4] `http://openjpa.apache.org/`

they are compiled to an intermediate format (the runtime readable form of the execution engine) that is called *ODE Object Model* and stored as .cbp files directly on disk.

Instead of relying on a separate message-oriented middleware system, Apache ODE has implemented its own, MOM-like system called Java Concurrent Objects or JACOB for short. The main goal of JACOB is to provide a persistent and asynchronous programming model that avoids the need to block execution while waiting for operations to complete. It furthermore allows to transactionally persist execution state for robustness and recovery.

Java Concurrent Objects (JACOB)[1]

- JACOB provides a persistent and asynchronous programming model for application-level concurrency. It does not make use of (expensive) operating system threads; it provides its own task abstraction.
- JACOB provides persistence of execution state via DAOs.
- Tasks in JACOB live in the execution queue which represents the current state of execution. Tasks may create other tasks, which are then appended to the execution queue.
- Tasks in JACOB communicate between each other through channels. Note that these channels are not queues in the sense of MOM; they are implemented as a java class, sending a message means invoking a method on that class.

3.2.3 ActiveBPEL

ActiveBPEL[2] is the open-source variant of ActiveVOS, the flagship product of Active Endpoints, the company behind both products. Besides supporting BPEL, the ActiveBPEL engine also implements people interactions by supporting BPEL4People and WS-Human Task.

Figure 3.3 depicts the architecture of ActiveBPEL. The ActiveBPEL engine

[1] `http://ode.apache.org/jacob.html`
[2] `http://activevos.com/community-open-source.php`

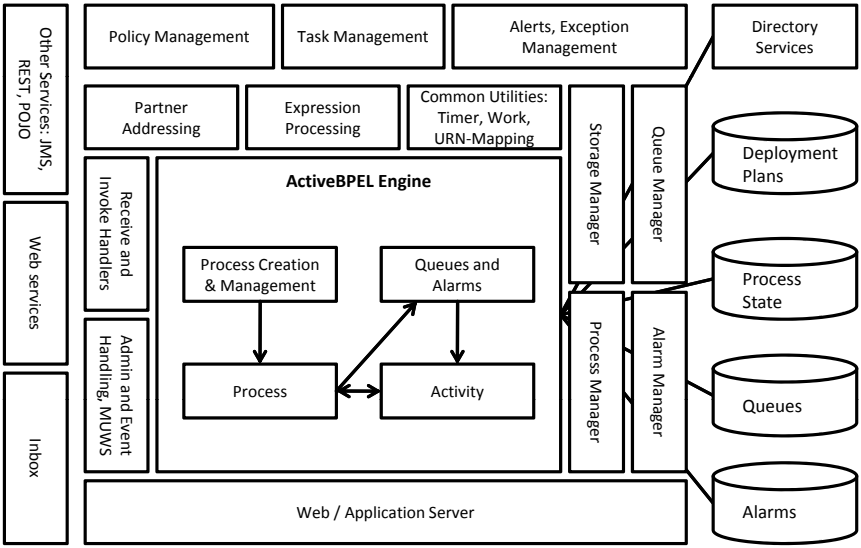


Figure 3.3: Architecture of ActiveBPEL (from the presentation "ActiveVOS - Server Architecture", available at `http://activevos.com/indepth.php`)

builds the core of the system and is presented in the center of the figure. It uses the concept of so called *managers*, components that perform operations on behalf of the engine [Wut06]. Managers are abstracted into interfaces, allowing to provide alternative implementations of their functionality, e.g. to be able to change the implementation of the job scheduler or the persistency layer. Examples for managers are: the *process manager* that is responsible to create process instances and to manage their state, the *queue manager* that manages receive queues and correlations, the alarm manager that provides an implementation of a job scheduler and the storage manager that abstracts from the persistent storage used, such as a certain RDBMS.

Application Server

- Through Apache AXIS2 deployed on Apache Tomcat, functionality to

implement Web service endpoints (e.g. for `receive` and `pick` activities) and also Web service client functionality used to implement e.g. the BPEL `invoke` activity is provided. Similarly, ActiveBPEL's administration API is exposed as Web services using the provided Web service runtime.
- The administration console of ActiveBPEL uses the provided Servlet and JSP implementation.
- The application server also manages the lifecycle of ActiveBPEL, i.e. it controls startup and shutdown of the engine.

The Database Manager provides implementations for the database systems IBM DB2, MySQL, Microsoft SQL Server and Oracle. Also, an implementation for a native XML database, Software AGs Tamino, is provided. However, persistent operation is optional in ActiveBPEL. When in *in-memory* mode, the engine does not use a database. The following list therefore assumes that ActiveBPEL runs in *persistent* mode.

Relational Database Management System

- The database system is responsible to persist execution state of the engine through serializing the contents of the execution queue.
- It is responsible to persist process instance state by serializing it to XML and then storing it in the database.
- The database acts as a persistence back-end for the receive queue. The state of active Web service endpoints waiting for a message to arrive is persisted through writing MessageReceiver objects to the database.
- All Database actions are executed under local transactions.

ActiveBPEL does not explicitly employ a message-oriented middleware to implement the queues mentioned before. Instead, a queue in ActiveBPEL is a synchronized list that can be serialized to the database. The major advantage of this approach is that distributed transactions are not necessary: persistence is solely implemented by the database system. A local transaction therefore suffices to handle requests to persist engine state (through the execution queue),

process instance state and receive queue state.

Usage of Queues (via Queue Manager)

- A queue is used to implement the execution queue of the engine. Two versions are available: the in-memory queue stores all entries in volatile memory only. The persistent execution queue in contrast is backed by a database and means are in place to handle failure to recover from crashes.
- A queue is also used to implement the Web service receive queue that buffers incoming Web service calls, correlates them to a running process instance or starts a new process instance. Equal to the execution queue, the receive queue also comes in two versions: in-memory or persistent.
- The reply queue buffers replies to synchronous Web service requests and therefore provides input for the Web service receive handler.

3.2.4 Discussion of Results

All discussed workflow systems use between two and three middleware systems as basis for their implementation. Every system uses a RDBMS for transactional persistence and expects an application server to provide a runtime environment that manages the engine's lifecycle and provides standard services like a Servlet, JTA or JNDI implementation. Some workflow engines directly use a messaging middleware, others implement MOM-alike functionality in form of custom components. Common to all is the use of different kinds of middleware systems as the basis for their implementation and to fulfill the requirements discussed in Section 3.1.1 and 3.1.2.

Most notably however, in all cases a persistent queue builds the basis for the implementation of the navigator component – the central part of a workflow engine governing the execution of activities the workflow is composed of. SWoM directly uses a persistent queue from the underlying MOM, ActiveBPEL uses a persistent, synchronized list and Apache ODE uses the JACOB runtime that also has a custom implemented, persistent queue at its heart. Also, the general setup of queues is very similar in all engines: mostly, three queues are

used, one for the navigator (execution queue), another one to buffer reception of Web service messages (receive queue) and a queue to buffer answers to Web service requests (reply queue).

In is very interesting that not only the open-source workflow engines we examined do all follow a similar architecture. IBM's FlowMark[1] [LR98] for instance is a commercial WfMS that has a very similar architecture based on RDBMS and messaging middleware. Here, four persistent message queues are used as the communication channels for clients, the program execution engine, the administration service and the navigation engine. Furthermore, the navigator and the administration service operate on a RDBMS and employ distributed transactions between the RDBMS and the messaging system.

The reason why the navigator component is architected very similar in all reviewed engines is QoS: if the navigator would be implemented using programming language features only to schedule activities of running process instances, all state would be lost after a crash. The reason is that the call-stack, i.e. the piece of memory that is used to keep track of running subroutines and the points where control should be returned after they are finished, is not persistent and thus the information on where the program exactly was when the system crashed is lost. In order to prevent this and to provide a "persistent call-stack", the navigator of a workflow engine follows a completely different design: instead of relying on the call-stack, activities to be executed are abstracted into actions that are added to a queue (the *execution queue*). The queue itself is persistent, i.e. stored on non-volatile memory. The navigator component runs an infinite loop that (i) consumes an action from the queue, (ii) executes it and (iii) appends subsequent actions to the queue. If an action needs to be executed multiple times (e.g. activities within a `while` block), they have to append themselves to the queue again. From an external viewpoint, the program representing the navigator component contains a second, "flat" call-stack in the form of the execution queue. Figure 3.4 visualizes this scenario.

This figure also shows a second, equally important feature of the navigator: since all actions of the engine (i.e. actions from all running process instances)

[1] A predecessor of IBM MQSeries Workflow which later evolved into IBM WebSphere Process Server.

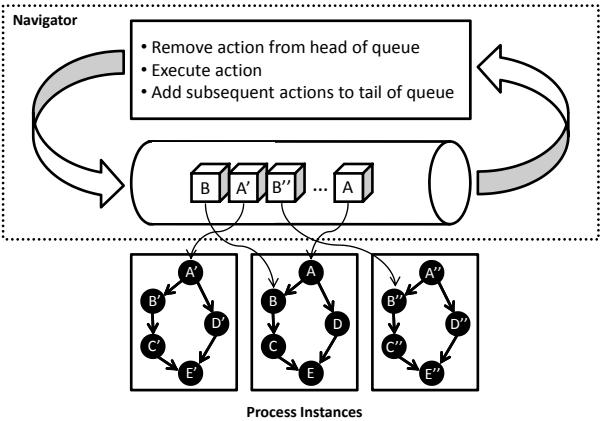


Figure 3.4: Abstract Architecture of a WfMS Navigator

are queued in the execution queue, it provides "application-level" concurrency. Actions from different process instances are interleaved, providing concurrency from an application point of view, but without using expensive operating system resources such as threads.

The fundamental problem with this design of a WfMS navigator is that the execution queue is an inherently logically central element, making the navigator component itself a central point of control in the engine. The execution queue and thus the middleware that provides it, is the key item for decentralization. In order to design a decentralized WfMS, the execution queue needs to be replaced by a means that does provide similar features, yet not introduce a single point of control.

The solution to this problem presented in this thesis is to employ *token passing* between distributed token buffers implemented by tuplespaces instead of MOM. Details on the token passing mechanism and its theoretical foundation are presented in Chapter 4. This chapter is focused on middleware for workflow systems, we thus continue with an analysis of the differences of MOM as the class of middleware currently used to implement the execution queue of a WfMS, and investigate if tuplespaces are a viable alternative.

3.3 Comparison of Tuplespaces and MOM

MOM has been compared to other middleware systems before. In a position paper for the HPTS 95 workshop, Jim Gray compares message queueing with database systems and concludes that "queues are "interesting" databases" [Gra95]. He suggests that the `enqueue` operation is similar to a sequence of `insert` and `commit` in a database and a `dequeue` is a transaction that includes message removal, message processing and the addition of the message to another queue followed by a `commit`. The database system in this case needs to be able to support large numbers of small transactions and a special transaction isolation level on queues in order to allow at most one removal operation at a time, e.g. a specialization of the *serializable* isolation level known from traditional database systems. [GR93] goes as far as suggesting concurrency control mechanisms to implement queues with a database system, e.g. in the form of the operations `Read_Past`, `Read_Through` and `Notify`. He further strengthens this argument by observing that the database manager takes care of tasks like creating checkpoints, transaction coordination, querying, security, locking, logging, recovery, utilities, performance monitoring, etc. and thus essentially the same tasks that are carried out by queue managers in message queueing systems.

Similarly, there have been attempts to declare queues and tuplespaces as essentially being equal [Ley06, FKLT07]. In this section, we revisit the arguments made and contribute a new view to the discussion by applying EAI, the primary area of application of MOM, to tuplespaces. The comparison is split into two parts: in Section 3.3.1 we compare both middleware systems based on their features only, which is then followed in Section 3.3.2 by an in-depth comparison of both on the basis of enterprise integration patterns.

3.3.1 High-Level Comparison of Features

After its definition by Gelernter in his work on the Linda programming language [Gel85], tuplespaces have seen numerous extensions and implementations. It is therefore out of scope of this thesis to give a precise definition that includes

all features. Enumerating features however is not the goal of this section. We rather want to discuss the core concepts behind both middleware systems and compare them with each other. These core concepts are captured by the initial proposal of the Linda coordination language and some commonly used extensions like advanced matching, leases, transactions and push operations (a.k.a. notification).

In the following, we will refer to Linda and Gelernter's work on tuplespaces e.g. [Gel85, GC92] by using the terms "original system" or "original idea". When it comes to technical questions or comparisons, we will use JavaSpaces[1], a prominent specification from SUN for tuplespace implementations in Java, as the basis for the discussion.

Likewise, there are many different implementations of message-oriented middleware systems available today. Consequently, we distinguish between the "original concept" by referring to [BHR95]; when it comes to technical questions however, we will argue based on JMS, a popular Java interface and specification for message queueing. The most basic concept in MOM is that of a *channel*, which is often further detailed in point-to-point or publish-subscribe channels, indicating two fundamental communication paradigms in MOM. A queue for instance is a specialization of a point-to-point channel indicating that an order is defined on the elements it contains.

Influenced by previous work in [FKLT07] and [Ley06], Table 3.1 presents a high-level comparison of both middleware systems.

One of the most notable differences is the architectural style these systems promote: tuplespace-based applications typically follow a blackboard architecture [BMR+96] where clients share data and coordinate themselves by accessing a shared piece of memory. Messaging-based applications in contrast typically follow a pipes-and-filter architecture [BMR+96] where an application is structured into components (*filters*) that are connected using messaging channels (*pipes*).

In the original tuplespace model, data is actively pulled from the space i.e. clients have to explicitly invoke a `read` or `take` operation in order to

[1] `http://www.jini.org/wiki/JavaSpaces_Specification`

Table 3.1: Feature-based Comparison of Tuplespaces and MOM

Differentiator	**Tuplespace**	**MOM**
Architectural style	Blackboard	Pipes and Filters
Pull vs. Push	Originally, only support for pull	Equal support for both push (e.g. `onMessage` callback) and pull (via `receive`)
Multiple read	Naturally supported	Not part of the original concept
Defined order on data items	No order	Order imposed by channel
Random access to data	Yes	No
Directed communication	Undirected	Unidirectional by definition
Data structure	Tuple	Message

access data. Although this has been addressed in later versions by introducing means for notification, the only way to access data in the original model is for client applications to invoke a method of the tuplespace API (or indirectly via the coordination primitive used). A corner case are the blocking versions of `read` and `take` since the client thread is suspended and woken up by the middleware if a matching tuple is available. Wake-up however requires a previous call to a `read` or `take` operation and ends this operation – the client is not continuously notified when matching tuples are available. Push and pull operations are native parts of the API of a MOM. In JMS for instance, push support is implemented by the `onMessage` callback and pull via the `receive` method.

As noted in [Ley06], in the original messaging model it is not possible to read the same message multiple times from a point-to-point channel, which is naturally supported using the `read` operation in tuplespaces. This deficiency has however been acknowledged and a `read` operation (see e.g. the JMS QueueBrowser) has been introduced to messaging middleware.

Another notable difference is the absence of an order on the items stored in a tuplespace; each client has equal access to all available tuples. A defined order on items however is one of the core principles of point-to-point channels in messaging. Because of that order, it is always clear which item is accessed when issuing a `receive` operation on a queue. In tuplespaces however, subsequent read operations on an unchanged tuplespace may return different tuples. This is known as the "Linda multiple read problem" [RW96], a problem that, by definition, is unknown to messaging middleware since the order uniquely identifies the element to be read next.

Related to this is the ability of tuplespaces to access data in random order, whereas in MOM the order of messages buffered by the channel also defines how these are accessed. A noted before, this has been solved by introducing a read operation e.g. in the form of the JMS QueueBrowser.

By definition, a channel in MOM always has a direction. For two-way communication, e.g. a request-response interaction, a separate channel for each direction is required. This stays in contrast to tuplespaces, where according to [Gel85]: "each tuple is equally accessible by each client and bound to none". Clients may access tuples independently of each other and thus read and write tuples as required. There are no restrictions on whether only reads or writes are allowed.

Another important difference are the data structures used by both systems: tuplespaces employ tuples (an ordered list of types fields) as means to structure the data stored into a tuplespace. Messaging middleware instead employs messages that do not mandate any particular data structure (JMS for instance supports BytesMessage, MapMessage, StreamMessage, ObjectMessage and TextMessage). However, messages always contain a *header* and a *body* part; the content of the header is addressed to intermediary nodes, the content of the body is addressed to the ultimate receiver of the message.

Besides theses differences, tuplespaces and messaging systems also have common properties, which are outlined below.

Communication middleware Tuplespaces as well as messaging systems are communication middleware; their intend is not to offer database func-

tionality such as support for complex queries. Instead, their common usage is characterized by providing communication facilities for short-living messages. As such, many implementations of both systems for instance support the attachment of an expiration date or lifetime indicator to a message, called *time-to-live* (TTL) in messaging or *lease* in tuplespaces.

Decoupled communication Both middleware classes employ a third party that buffers the communicated messages in order to be able to tolerate unavailability of sender and receiver. Additionally, this communication buffer typically holds a virtual identifier (e.g. a queue name, a topic name or the name of a tuplespace) effectively decoupling sender and receiver by using an indirect address.

[AvdADtH05] analyzes the decoupling dimensions in a more formal manner using Colored Petri Nets. The authors conclude that both, messaging and tuplespaces provide the exact same decoupling properties. [Ley06] revisits the discussion on decoupling dimensions and finds that, given a particular type of messaging channel (non-persistent publish-subscribe), there is indeed a difference to tuplespaces. In this case, time decoupling is not given. Generally however it is agreed that both middleware systems decouple sender and receiver of a message in the dimensions: (i) time, (ii) reference and (iii) location [AvdADtH05].

Coordination capabilities Both systems offer similar coordination capabilities with respect to the semantics of the operations offered. In tuplespaces, operations exist in blocking and non-blocking variants, as well as notification facilities (in later systems). Similarly, messaging applications support blocking (e.g. JMS `receive`), non-blocking (e.g. JMS `receiveNoWait`) and notification (e.g. through JMS `MessageListener.onMessage()`) operations.

As noted in [FKLT07], both technologies can even simulate each other to a certain extend. In the following, we briefly outline the ideas how messaging

systems can be implemented using tuplespaces and how tuplespace semantics may be implemented using messaging middleware.

Tuplespace → MOM (point-to-point) Figure 3.5 presents a solution on how to implement a point-to-point channel with tuplespaces. Figure 3.5(a) presents a queue that holds (from head to tail) the messages *A* through *G*. In order to preserve the order of messages, an *index tuple* (gray background color) is used in the tuplespace implementation presented in Figure 3.5(b). This tuple holds the IDs of the tuples that currently are at the head (index 1) and tail (index 7) of the queue. A client that wants to remove a message from the queue first reads the current ID of the head, then removes the corresponding tuple and finally increases the head ID of the index tuple. Similarly, if clients want to add a tuple to the queue, they first read the current tail ID from the index tuple, increase it by one, write the resulting number back to the index tuple and write a tuple to the space with an index pointing to the new tail ID.

Problematic in this implementation is the index tuple that is accessed two times (by a `take` and `write` operation respectively) for each access of a message in the tuplespace.

Tuplespace → MOM (publish-subscribe) With the non-destructive read operation, tuplespaces offer a way to communicate one tuple to many clients. Extending this simple read operation to a publish-subscribe solution however requires subscriptions and mechanisms to structure the messages communicated (a.k.a. topic trees) to be implemented by application logic on top of the tuplespace. Similarly, a solution is required to clean-up tuples after all clients have received them.

One possibility is to maintain a tuple that records all clients that are interested in a certain topic and let them acknowledge the reception of each message. A tuple then must only be removed if all registered clients have acknowledged its reception. This solution however suffers from the same problem as the tuplespace-based point-to-point channel implementation: the index tuple where all clients record the acknowledgment

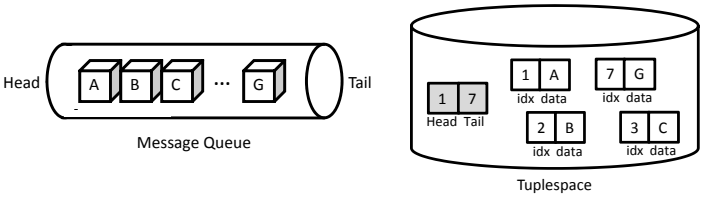
of reception of each tuple is the bottleneck of the system. As a result, complex and expensive client-side logic is required to implement the desired semantics.

MOM → tuplespace MOM systems implementing the JMS standard already support one of the core functionalities of tuplespaces: *content-based addressing*. The Message Selector expression is applied to all messages in the channel, non-matching messages are skipped and matching messages are returned to the client. Together with the usual MOM operations on channels, the functionality around a JMS Selector is roughly equivalent to what is provided by tuplespaces. A selector may be used for destructive and non-destructive read operations on point-to-point and publish-subscribe channels, with blocking and non-blocking operations and also for notifications.

It is important to note however that the Selector does not provide a full equivalent to tuplespace functionality since it can only be applied to the header part of a message whereas a template in tuplespaces is applied to all fields of a tuple.

In fact, many implementations of tuplespace or messaging systems borrow ideas from both worlds: as noted before, JMS Selectors for instance implement content-based addressing, which is a core tuplespace concept. Similarly, the read operation in form of the JMS QueueBrowser was already part of the original Linda coordination language. Notification on the other hand is a concept that was introduced in messaging systems first, but then also occurred in tuplespace implementations.

In the following section, we present a comparison of both systems from a completely different angle. Rather than looking at features, we discuss the application of tuplespaces in the area of enterprise application integration (EAI), the primary area of use for messaging middleware.



(a) Queue

(b) A Queue Implemented Using a Tuplespace

Figure 3.5: Simulating a Queue with a Tuplespace

3.3.2 Comparison Using EAI-Patterns

Enterprise applications consist of a large number of individual applications, often comprising roll-your-own applications, legacy applications, acquired standard applications, etc. running on multiple application servers and depending on various data sources. Because business users of enterprise applications do not use them in isolation but require an integrated and inter-operable use, a separate integration layer must be provided to result in the required business view of the application scenery. Also, enterprise application infrastructures grow and evolve over time, thus, the separate integration layer grows at least at the same pace. Building such an integration layer is typically referred to as creating an Enterprise Application Integration (EAI) solution.

When creating an EAI solution, one often recognizes certain recurring problems and that the same actions are taken over and over again to solve these problems. Recurring problems and their solutions can be described as *Patterns* [A+79]. Besides the well-established patterns in the field of object-oriented software design (e.g. [GHJV95]), there is an emerging area for creating patterns for software and enterprise architecture (e.g. [Fow02, BMR+96]). [HWB03] defines a set of patterns covering prominent examples of situations encountered in message-based EAI solutions.

In [MWSL07], we present a comparison of messaging and tuplespace technology by investigating their application in the field of EAI. For this purpose, we

provide a comprehensive analysis of the landscape of EAI patterns presented in [HWB03] by assuming tuplespace technology rather than MOM as their underpinning. In the following section, we summarize our findings from the analysis presented in [MWSL07] and combine them with the results of the feature-based comparison from Section 3.3.1.

3.3.3 Discussion of Results

Table 3.2 summarizes key findings from [MWSL07] and Section 3.3.1. Interestingly, in the detailed comparison using EAI patterns [MWSL07], all generic differences listed in Section 3.3.1 are also covered – although in the context of EAI and not from the generic viewpoint as taken in Section 3.3.1.

The first significant difference we found in our EAI-pattern based analysis is the mismatch of data objects native to each technology. While tuplespaces rely on *tuples*, MOM promotes the notion of a *message*. It is not a syntactical mismatch however, since tuples and messages can be arbitrarily structured and mapping between both formats is straight-forward. The mismatch rather is on a semantical level: messages contain two designated parts: message header and message body. The header part contains information that is addressed to the MOM itself (e.g. to the intermediary nodes forwarding the message during transmission), whereas the body of a message is addressed to the ultimate receiver. Tuples do not distinguish between header and body, thus information targeted at the transmission middleware is intermixed with data targeted at the ultimate receiver.

A result of Linda's generative communication paradigm [Gel85] and the absence of (directed) channels is, that tuples exist in the tuplespace independently of any client application. They do not have a designated message path and they can be accessed in the same way by any client that has access to the tuplespace. This means that e.g. the publisher of a tuple can decide to retrieve this tuple again, modify it, and write it back to the space. This way of interaction with a tuplespace is not supported in MOM based application systems: a message, once sent to a channel, cannot be retrieved by anybody else than the client consuming messages from the channel. This is a consequence of the fact

Table 3.2: Most Significant Differences between Tuplespaces and Messaging [MWSL07]

Differentiator	**Tuplespaces**	**MOM**
Tuple vs. Message	Not aware of the semantics of a Message.	A message is the data structure used.
"Implicit Routing"	Spaces do not have the concept of routing.	Message routes are explicitly defined through channels at design time.
"Interception Problem"	It is not possible to define the order of concurrent read operations on the same tuple	Since all possible routes for messages are defined explicitly, this problem does not exist.
Original Point-to-Point/Publish-Subscribe Semantics	Not exactly reproducible with tuplespaces	Naturally supported by MOM technology.
"Incomplete History"	A client may miss interesting messages when being offline.	The Durable Subscription pattern allows clients to be temporary offline, but still ensures that messages are re-delivered when the client comes back online again.
Promoted architectural style	Blackboard	Pipes-and-Filters
Defined order on data items	No order	Order imposed by channel
Random access to data	Yes	No

that MOM channels are uni-directional. The client that sends a message to a channel cannot itself be a consumer on that same channel.

Messaging uses explicitly defined *Message Channels* to deliver messages from a sender to possibly multiple receivers (publish-subscribe channels, as opposed to point-to-point channels with exactly one receiver for a given message). As

applications that build upon messaging do typically consist of a network of various message channels, message routers are necessary to move messages from one channel to other channels based on information in either the message header, the message body or both. In tuplespaces, messages are not delivered from a sender to a consumer directly, but rather published to a shared data space by the sender and retrieved by the consumer via addressing using templates (cf. content-based addressing). A routing process with explicit routing rules as in messaging technology does not exist. Routing in tuplespaces is the internal process how a template is applied (e.g. using an index) and how the resulting tuple is handed back to the client. Also, tuplespaces are centered around the idea of a "shared data space", meaning that even the data flow itself is not visible anymore, but defined in an ad-hoc manner by the templates of the tuplespace clients taking part in the interaction.

A consequence of the data flow in tuplespaces not being explicitly defined is that debugging and auditing is much harder to achieve in such systems. Message routes are explicitly defined through channels in MOM-based applications, in tuplespaces however it is very hard to understand the route a tuple describes when it is consumed and written back by many different clients.

While the template-based retrieval operations of tuplespaces allow routing to be defined implicitly as discussed before, some patterns require being able to define an order in which consumers having subscribed using a similar template should be notified upon insertion of a matching tuple. For the implementation of e.g. the *Wire Tap* pattern, a "wiretap consumer" which non-destructively consumes a data object before the actual consumer consumes the object, needs to be specified. Since current tuplespaces implementations do not provide such operation ordering features but rather select the consumer to be notified non-deterministically, this functionality needs to be implemented on top of the middleware in order to have the same functionality at hand that MOM already provides. As a result, extra application logic is required in tuplespace-based applications to implement EAI patterns such as *Wire Tap* or *Detour*.

Both, publish-subscribe and point-to-point channels are first class citizens in MOM systems. The way of communication these patterns provide however does not fit to the way how clients interact with a tuplespace: it is a funda-

mental difference between directed communication over dedicated channels and anonymous publication and retrieval of data over a shared space. While directed communication can be implemented on top of tuplespaces by introducing "subscription tuples" as discussed in Section 3.3.1, this solution is still very inefficient compared to the messaging solution. Essentially, these problems come from the different properties of the underlying data structures used for storing the data in each system: multisets in tuplespaces as opposed to lists of messages in messaging.

The problem of "incomplete history" arises when taking a closer look at the Durable Subscriber EAI pattern. A MOM that supports this pattern allows subscribers to be temporarily offline and re-delivers all missed messages in the order they arrived when the client comes back online. In tuplespaces however, other clients may remove a tuple while the intended receiver is offline. The middleware provides no means to ensure that all matching tuples are delivered to a client when coming back online.

3.4 Comparison of Tuplespaces and RDBMS

As discussed in Section 3.2, relational database management systems (RDBMS) are another frequently used middleware together with MOM to build workflow systems. Although being focused on persistence, database management systems share some properties with tuplespace middleware, which are analyzed in this section.

Table 3.3 summarizes the main differences of relational database management systems and tuplespace middleware and begins by pointing out the differences of both middleware systems in their general area of application: tuplespaces are typically used as communication middleware, as a means for applications to coordinate their actions with others by writing and reading tuples from a shared space. Relational database systems in contrast are used to persistently store data and provide means to run complex analysis using a rich query language.

Providing support for data analysis tasks is not focus of tuplespace middleware, hence, very simple means for identifying the tuple to be read are

Table 3.3: Most Significant Differences between Tuplespaces and RDBMS

Differentiator	**Tuplespaces**	**RDBMS**
General Use	Communication middleware	Persistent storage and data analysis
Data Access	Very restricted query language	Sophisticated query language
Data Structure	Arbitrary structure of tuples in a single tuplespace allowed	Equal structured tuples per table.
Coordination capabilities	Linda + extensions	None

available. Typically, this is done using a technique called *template matching*: the client provides values for a subset of the fields of the tuples to be read. Interestingly, a very similar technique was developed for database systems, before the structured query language (SQL) [SKS98] became mainstream. It is called "Query by Example" (QBE) [Zlo77] and follows the exact same idea as template matching in tuplespaces.

Both systems rely on *tuples* as the data structure. While tuplespaces allow arbitrary structured tuples to be stored in the same tuplespace, relational databases allow only equal structured tuples to be stored in a table (a.k.a. relation). Each different structured tuple requires a separate table. The reason lies in the relational data model that builds the basis for relational databases.

Lastly, tuplespaces natively support coordination of applications through blocking operations, an aspect not supported as first class citizen by relational database systems.

Generally, a trend can be observed that tuplespaces adopt selected features from database systems to complement tuplespace functionality. The resulting systems are essentially hybrid, covering tuplespace functionality but also offering basic operations typically found in database systems. A prominent example for this trend is IBM's TSpaces[1]. Apart from basic tuplespace functionality, it

[1] `http://www.almaden.ibm.com/cs/tspaces/`

offers operations: (i) to update tuples (ii) transactions, (iii) persistence with journaling and checkpointing for crash recovery, (iv) indices and (v) complex query expressions like nested AND or OR queries, support for XPath and regular expressions.

3.5 Conclusion

The focus of this chapter was on middleware systems used to build workflow systems. We started by introducing a list of operational requirements on production workflow systems and concluded that a proper architecture and using middleware systems such as an RDBMS and MOM supporting the chosen architecture are the key items to fulfill these requirements.

The findings were validated by reviewing the architecture of three open-source BPEL engines and highlighting the role and use of middleware in their architectures. The review has shown that the most commonly used middleware systems for workflow engines are application servers, MOM and RDBMS.

Subsequently, we discussed the requirements on the execution model of a decentralized workflow engine and concluded that the traditional workflow engine design – using a central database to persist state and a central, queue-based navigator component to flatten and persist the call stack of running process instances – is not suitable to implement a decentralized engine. Instead, we propose an execution model based on *token passing* which is implemented by middleware that provides decoupling for each individual component in the dimensions of time, reference and location.

As both, tuplespaces and MOM provide these decoupling dimensions, we continued with a detailed analysis of the differences of these systems. The analysis was structured into two parts: first, both systems were compared based on their features. The second part approached the comparison from another angle. EAI is the primary area of application of MOM; in the comparison, we assume tuplespaces instead of MOM as the middleware to implement commonly used EAI patterns.

As a result of this analysis, we found that:

- Messaging and tuplespaces promote different architectural styles: while messaging middleware is typically used to implement pipes-and-filter based architectures, tuplespaces promote the blackboard style.
- A defined order is a major characteristic on the messages delivered by message oriented middleware. Tuplespaces in contrast explicitly allow "independent", i.e. random access to the tuples stored in a tuplespace.
- Messages in MOM travel through explicitly defined routes (a.k.a. *channels*) whereas routing in tuplespaces is "implicit".
- While the syntactic mapping between messages and tuples is straightforward, the difference is on the semantic level. Tuples do not distinguish between header and body, i.e. between data addressed to the middleware system and to the ultimate receiver.
- Tuplespaces do not have an equivalent counterpart of a *Durable Subscription*: tuplespace clients may miss tuples they are interested in while being offline, their "historical knowledge" about the system is incomplete.
- Tuplespace clients cannot define the order of concurrent read operations on the same tuple, a fundamental requirement to implement i.e. the *Detour* or *Wire Tap* EAI patterns that allow to intercept a given message flow.
- The original point-to-point and publish-subscribe channel semantics are not exactly reproducible using tuplespaces, but naturally supported by message-oriented middleware.

Next, we discussed similarities of tuplespaces and relational database management systems. Although these middleware systems have fundamentally different backgrounds (i.e. data storage and analysis vs. communication middleware), relational database management systems have influenced the development of tuplespace middleware. Examples are the adoption of a richer query language, mechanisms for persistence and recovery of the stored tuples, transactions, operations to update tuples, indices, etc. Some tuplespace systems can be already be seen as hybrids, offering basic database functionality together with the implementation of tuplespace specific operations such as template-based retrieval of tuples and blocking operations. Sometimes, an

RDBMS is even used as a basis for the implementation of a tuplespace[1].

At the same time, the comparison to MOM showed that tuplespace systems can also implement many of the patterns typically implemented using messaging systems. A similar trend can be noticed here: there are tuplespace systems that adopt MOM features such as ordered semantics instead of multiset semantics for tuplespaces[2].

In essence, a tuplespace may implement basic features from both, MOM and RDBMS. The preference which feature set is emphasized highly depends on the envisioned area of application of the tuplespace implementation. Figure 3.6 visualizes these findings and presents an abstract spectrum showing tuplespaces in the "feature center" of MOM and RDBMS. Depending on the set of features provided, a concrete tuplespace implementation may lean more towards messaging or RDBMS.

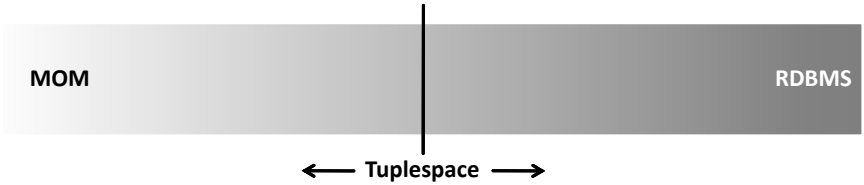


Figure 3.6: Tuplespaces as the Middle of the Spectrum between MOM and RDBMS

As a result, tuplespaces can be seen as a concept that unifies features of two of the most commonly used middleware systems to implement workflow engines. Features adopted from messaging middleware help to implement the navigator component and the decoupling of the components that communicate using tuplespaces. Features from RDBMS help to implement non-functional properties such as availability or reliability through transactions and persistency and provide basic data management functionality such as tuple update operations or a richer query language.

[1] Earlier versions of MozartSpaces (`http://www.mozartspaces.org/`) for instance.

[2] See e.g. the `config.setOption(ConfigTuple.FIFO, Boolean.TRUE);` directive in TSpaces or the different Coordinator types supported by MozartSpaces.

A very important point is that tuplespaces provide a natural way to implement the principle of token passing – the core idea behind the decentralized process execution model developed in the next chapter. Tuplespace-based coordination is an implementation of this principle, providing the means for decoupled components to coordinate themselves by writing and reading tuples from shared spaces.

Chapter 4

Distributed Process Execution Model

This chapter introduces a novel, Petri-net based execution model for the decentralized execution of workflows. It is based on tuplespaces [Gel85] as the underlying execution middleware, and models their capabilities closely. It facilitates decentralized execution of process logic by employing token passing as the mechanism for the *coordination* of distributed navigation components. The focus of our model is workflow *execution*; its underlying formalism is thus called Executable Workflow Nets (EWFN) [MWL08a, MWL08b]. The central property of EWFNs – inherited from Petri nets – is the absence of a central point of control: each transition only depends on directly preceding and directly succeeding places; it is decoupled from the remainder of the net.

We use EWFNs as a formal metamodel for tuplespace-based interactions. EWFNs are, on the one hand directly executable on a tuplespace system and thus play the role of "microcode" within our workflow system. On the other hand, EWFNs have a graphical representation and thus allow for modeling tuplespace-based, distributed applications – distributed workflows essentially

are specializations of such applications. Consequently, EWFNs build the groundwork for the main contribution of this thesis, the decentralized workflow execution engine. In the form of patterns, EWFNs can be the basis for the execution of higher level workflow languages such as WS-BPEL [A+07] or BPMN [W+09]. How BPEL is covered is shown in Chapter 5.

EWFNs fill two important gaps in state of the art research: (i) they present a fundamentally new way to execute workflows by decentralized *coordination* of fully decoupled, individual navigation components instead of employing central *navigation*. This allows for arbitrary deployment of partitions of the workflow graph and thus flexibility in the deployment of process models. Additionally (ii), they define a formal model and a graphical language for the expression of tuplespace-based interactions and can thus be used to graphically represent tuplespace applications on the one hand and to apply verification techniques to (EWFN modeled) tuplespace-based applications on the other hand.

This chapter is structured as follows: after providing formal descriptions of the syntax and semantics of EWFNs, advanced concepts such as conflict detection, join matching and structural properties are covered. We present a graphical notation for EWFNs, which is subsequently used to provide an alternative formalization of the basic workflow control flow patterns documented in [RtHvdAM06]. By creating this formalization, we emphasize on the applicability of EWFNs to workflow definition languages. We furthermore present a mapping of EWFNs to Place-Transition Nets (PT-nets) [Rei85] and thus connect EWFNs to the world of Petri net formalizations allowing existing Petri net theory and algorithms to be applied to the PT-net version of a given EWFN.

Please note that, since tokens in EWFNs are tuples, the words “tuple” and “token” are used interchangeably. The term EWFN without subscript is used when no distinction between the different types of EWFNs is required or the particular type of EWFN is clear from the context.

4.1 Motivation

The close relationship between tuplespaces and Petri nets is well known in research. Petri nets are sometimes even seen as a predecessor of Linda tuplespaces: both represent state as a multiset of tuples and postulate the use of `give` and `take` operations instead of `read` and `write` [BR06]. In this spirit, we continue the initial thoughts on the similarities of Petri nets and Linda and develop a Petri net dialect that is directly executable on tuplespaces. This way, both approaches are in a model-implementation relationship with each other: Petri nets are the formal underpinning providing a precise metamodel for the implementation of the tuplespace middleware used for execution.

Note that building such kinds of relationships is common practice in information technology, e.g. in the field of relational databases. There, relational algebra and relational database management systems are in similar relation to each other.

Our motivation to provide a rigorous formal approach for tuplespace-based interactions is two-fold: (i) Petri nets are an accepted, well understood formalism. In fact, they are one of the most prominent formalisms used in workflow research. (ii) It is important to have an abstract notation that enables a clear separation between model and implementation, i.e. between the high-level concept and implementation issues.

4.2 Executable Workflow Nets

Petri nets were originally designed as a model for an arbitrary extensible computer architecture i.e. machines that consist of many individual modules, each of them responsible for a particular task of the overall system. Adding a new module should have no impact on the existing ones; their performance characteristics for instance, must not change [Rei85]. Three underlying design principles facilitate this behavior: (i) there is no central point of control, e.g. there is no central clock. (ii) Each action is triggered locally, and has only local effects. (iii) Petri nets are inherently asynchronous in nature and communication solely happens over local communication via places connected

to transitions.

These principles build the foundation for our model, that is naturally based on Petri nets. In their spirit, we define a set of individual components and the communication between them. The communication middleware that facilitates component interaction during execution of the model is based on tuplespaces, since (i) they closely resemble the design properties of Petri nets in terms of loose coupling and asynchronous communication [AvdADtH05] and (ii) each element of a Petri net can be directly mapped to an entity in a tuplespace based system (either a program, a tuple or a tuplespace).

At the most basic level, a tuplespace provides a buffer for storing and retrieving tuples (destructively and non-destructively). Analogously, places in Petri nets provide buffers for synchronizing interactions between transitions. The structure of tokens can vary depending on the employed Petri net dialect: while the original model of Petri nets [Pet62] does not distinguish between different kinds of tokens (i.e. there is only one "kind" of token), Colored Petri Nets (CP-Net) [Jen92] are a widely used extension allowing to distinguish between tokens according to an assigned type – very much like the tuple model in tuplespaces allows tuples to contain any number of typed fields. Using such a type model for tokens in Petri nets therefore increases similarity between both models.

Actual processing of computation logic in tuplespace-based applications is carried out by tuplespace-external applications that specify the conditions under which they can commence their execution in form of a template on tuples available in the tuplespace. Similarly, the execution (firing) of a transition depends on the presence of tokens in its input places. In Place-Transition Nets (PT-nets), the start condition of a transition is defined as at least one token being present in each input place of the transition. CP-Nets, in contrast, allow for attaching complex expressions on token type and value on both incoming and outgoing arcs of transitions, resulting in similar expressivity as Linda templates with respect to defining start conditions for tuplespace clients.

Once the start condition of a transition in a Petri net is satisfied, the transition destructively consumes a number of tokens from each input place; in CP-Nets the number of consumed tokens is dependent on the expression defined

on the respective input arc of the transition. In addition to this model of purely destructive token consumption in Petri nets, tuplespaces support both destructive (removing the tuple from the space) as well as non-destructive (leaving a copy of the tuple in the space) reception of a tuple. The need for such an operation has also been recognized in Petri net research, leading to the addition of read (a.k.a. test) arcs to the Petri net model [VSY98].

Both extensions together create a Petri net dialect that can be natively executed by a tuplespace system, i.e. each element in the Petri net model has an equivalent at the tuplespace side. Places in Petri nets are containers that can hold any number of typed tokens and can therefore be represented as tuplespaces. Tokens can be stored in places and consumed again by transitions, which effectively are tuplespace client applications. Tokens are represented as tuples communicated over the tuplespace. Operations possible with the resulting Petri net dialect are *writing* to a place, i.e. transitions that produce tokens and *taking* or *reading* tuples, i.e. destructively or non-destructively consuming a token from a place. Note that these three operations exactly match the basic operations of the interfaces exposed by virtually any tuplespace implementation available today. Thus EWFNs are *directly executable* on tuplespaces.

A direct consequence of this close relationship is that the described Petri net dialect can also be used as a modeling language for tuplespace applications. Users can model applications using EWFNs, and then use the created models as the basis for implementing the tuplespace-based application.

4.2.1 Syntax

To support both use cases – Linda modeling and workflow execution – EWFNs are defined in a modular manner. In the course of this work, three versions will be defined that cover the spectrum from the very generic Linda set of operations to various Linda extensions up to a specifically designed version to support the WS-BPEL process execution language. Figure 4.1 shows how these EWFN versions relate to each other. We start by defining EWFNs based on the original Linda set of tuplespace operations and semantics. This (basic) type of EWFN is called "$\text{EWFN}_{\text{Linda}}$" and – since Linda operations are supported by virtually all

available tuplespace implementations – this is the most generic type of EWFN. Linda however has seen numerous extensions since its original publication, later in this chapter we thus provide a second definition of an extended EWFN that includes four of the most prominent extensions to the Linda model. This type of EWFN will be referred to as "$\text{EWFN}_{\text{Extended}}$". Finally, in Chapter 5 an EWFN version called "$\text{EWFN}_{\text{BPEL}}$" will be presented that is used to formalize WS-BPEL. This EWFN type concentrates on the tuple structures required to formalize BPEL interactions, e.g. to distinguish between data and control flow or between positive and negative control flow for Dead-Path-Elimination [LR00]. We foresee that the EWFN definitions are extensible, e.g. to include another workflow language. The corresponding EWFN type can e.g. be created based on $\text{EWFN}_{\text{Extended}}$, as depicted by the $\text{EWFN}_{\text{Other}}$ element in Figure 4.1.

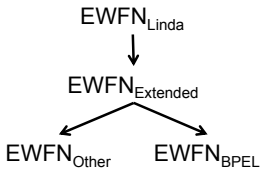


Figure 4.1: Different Kinds of EWFNs and their Relation to Each Other

The most often used data structure to represent a tuplespace is the *multiset*. In the following, we introduce multisets as far as needed in this work, a complete coverage of the subject matter can be found in [Jen92] or [Syr01].

A multiset (a.k.a. "bag") is a set that allows multiple appearances of the same element, i.e. the requirement on sets to only contain pair-wise different elements is relaxed. It is defined *over a set* S where the elements of the multiset are taken from. A multiset defines how many times a particular element is present in set S.

To indicate that a certain set is a multiset, we use superscript ^{MS}, e.g. set S is a multiset if we write S^{MS}.

Definition 1 (multiset)**.** *A multiset* M^{MS} *over a non-empty set* S *is a function* $M^{MS} : S \rightarrow \mathbb{N}_0$ *where the non-negative integer (including 0)* $n = M^{MS}(s)$, $s \in S$ *denotes the number of appearances of* s *in the multiset. An element* v *is a member*

of the multiset M^{MS} iff $M^{MS}(v) \neq 0$. $\emptyset$ denotes the empty multiset.

A set can be represented as a multiset, i.e. a set is a multiset with $\forall s \in S : M^{MS}(s) = 1$. Following the conventions in [Jen92], we represent the multiset M^{MS} as a formal sum:

$$\sum_{\forall s \in S} M^{MS}(s)`s$$

The multiset $\{a, b, b, c, d, d, d\}$ for instance is represented by the sum $1`a + 2`b + 1`c + 3`d$.

Definition 2 (addition)**.** *Let M^{MS} and N^{MS} be multisets over the set S, the result of the addition of $M^{MS} + N^{MS}$ is defined as*

$$M^{MS} + N^{MS} = \sum_{\forall s \in S} \left(M^{MS}(s) + N^{MS}(s)\right) `s$$

Example: Adding the multisets $\{a, a, b, c, c\}+\{a, d\}$ results in $\{a, a, a, b, c, c, d\}$

Definition 3 (comparison)**.** *Let M^{MS} and N^{MS} be multisets over the set S, the following statements hold:*

$$\begin{aligned}
M^{MS} \neq N^{MS} &\text{ iff } \exists s \in S : M^{MS}(s) \neq N^{MS}(s) \\
M^{MS} = N^{MS} &\text{ iff } \forall s \in S : M^{MS}(s) = N^{MS}(s) \\
M^{MS} \leq N^{MS} &\text{ iff } \forall s \in S : M^{MS}(s) \leq N^{MS}(s) \\
M^{MS} \geq N^{MS} &\text{ iff } \forall s \in S : M^{MS}(s) \geq N^{MS}(s)
\end{aligned}$$

The following equation holds for instance: $\{a, a, b\} \leq \{a, a, a, b, c\}$

Definition 4 (subtraction)**.** *Let M^{MS} and N^{MS} be multisets over the set S and $M^{MS} \geq N^{MS}$, then the result of the term $M^{MS} - N^{MS}$ is defined as*

$$M^{MS} - N^{MS} = \sum_{\forall s \in S} \left(M^{MS}(s) - N^{MS}(s)\right) `s$$

Example: Subtracting the multiset $N^{MS} = \{a, b, c, d\}$ from $M^{MS} = \{a, a, a, b, c, d\}$ results in $\{a, a\}$. The subtraction is allowed since $M^{MS} \geq N^{MS}$ holds.

Definition 5 (cardinality). *The cardinality of a multiset M^{MS} over S is defined as*

$$\left|M^{MS}\right| = \sum_{\forall s \in S} M^{MS}(s)$$

Example: The cardinality $\left|M^{MS}\right|$ of the multiset $M^{MS} = \{a, a, a, b, b, c\}$ is 6.

Definition 6 (intersection). *The intersection of two multisets M^{MS} and N^{MS} over the set S is defined as*

$$M^{MS} \cap N^{MS} = \sum_{\forall s \in S} \texttt{min}(M^{MS}(s), N^{MS}(s))`s$$

Example: The intersection of $M^{MS} = \{a, a, a, b, b, c\}$ and $N^{MS} = \{a, a, b, d\}$ results in the multiset $\{a, a, b\}$.

Definition 7 (union). *The union of two multisets M^{MS} and N^{MS} over the set S is defined as*

$$M^{MS} \cup N^{MS} = \sum_{\forall s \in S} \texttt{max}(M^{MS}(s), N^{MS}(s))`s$$

Example: The union of $M^{MS} = \{a, a, c\}$ and $N^{MS} = \{a, b, c, c\}$ results in the multiset $\{a, a, b, c, c\}$.

To improve readability, angle brackets are used to denote tuples, parentheses are used as usual to e.g. denote arguments of functions or preferences in set theoretical operations. A tuple of three elements is thus denoted as $\langle a, b, c\rangle$ instead of (a, b, c).

Definition 8 ($\text{EWFN}_{\text{Linda}}$). *$EWFN_{Linda}$ is a tuple*

$$EWFN_{Linda} = \left\langle \Sigma, \Phi, P, T, F, X, A, B, I_0, L_{write} \right\rangle$$

- *Σ denotes the set of* tokens *in form of arbitrary structured* tuples. *Each token is prepended with a "header" tuple $h \in \mathbb{N} \times \mathbb{N}$, representing* ProcessID *and* InstanceID. *This allows to distinguish tuples representing state of instances of different process models as well as tuples from instances of the same process model when present in a single tuplespace.*

It is often necessary to represent arbitrary structured data items in business processes. In this case, we serialize structured information (e.g. in form of an XML-DOM [LH+04]) into nested tuples. Finally, Σ contains the "empty" tuple ϵ used to denote that actually no tuple is produced.

- *Φ is the set of* types. *Types are used to indicate the structure of the tuple written by a transition, by specifying the type of each field of the tuple, e.g. $\langle int, int, string \rangle$ indicates the type of tuples that contain two integer and a string field. Note that the concrete type system in use is intentionally left open since EWFNs may be used to execute processes defined in different languages. Depending on the execution language, the sets Σ and Φ need to be adapted according to the type system used. $EWFN_{BPEL}$ for instance uses XSD [T+04] and its formalization from [KML08] to define the set Φ.*
- *P is a finite set of* places
- *T is a finite set of* transitions *such that $P \cap T = \emptyset$.*
- *$F \subseteq F_{pt} \cup F_{tp}$, with $F_{pt} = (P \times T \times R)$, $F_{tp} = (T \times P)$ and $R = \{read, take\}$. F is a set of arcs known as the* flow relation. *The arcs correspond to classical Linda operations:* write *arcs go from transitions to places (i.e. are member of the set $T \times P$), whereas* read *and* take *arcs go from places to transitions, with arc inscription R denoting the type of arc.* Take *arcs are known from classical Petri nets (i.e. they destructively consume tokens from places).* Read *arcs (also known as* Test *arcs) [VSY98] in contrast allow a transition to non-destructively read a token from a place.*
- *X is a set of* templates, *i.e. tuples that may either contain a wildcard ($\star$) or a concrete value on each field.*
- *$A : F_{pt} \rightarrow X$ is a function that assigns a* template *to incoming arcs of a transition such that $\forall \langle p, t, r \rangle \in F_{pt} : A(p, t, r) \in X, r \in R$. Sometimes, we use A without the last parameter, as a shortcut to access the template assigned to an arc pointing to a transition. In these cases, it is not important whether the template is used in a* `read` *or a* `take` *operation.*
- *$B : F_{tp} \rightarrow \mathcal{P}(\{\langle t_1, \ldots, t_n \rangle \mid t_i \in \Phi, n \geq 1\})$ assigns a set of* tuple types *to outgoing arcs of a transition such that $\forall \langle t, p \rangle \in F_{tp} : B(t, p) \in$*

$\mathcal{P}(\{\langle t_1, \ldots, t_n \rangle \mid t_i \in \Phi, n \geq 1\})$[1]. *The tuple type is a tuple with members of set Φ as elements to indicate the type of tuples produced by a transition. Note that a transition may even write different types of tuples depending on internal state. An arc modeling a branch of an exclusive choice for instance may either write a tuple indicating that the branch was taken or the empty tuple (ϵ) if another branch was selected. In the graphical notation of EWFNs (Section 4.2.7), each member of the set of tuple types of a write arc is separated by "|".*

- $I_0 : P \rightarrow \mathcal{P}(\Sigma^{MS})$ *is an* initialization function *that assigns a multiset over Σ to places such that $\forall p \in P : I_0(p) \in \mathcal{P}(\Sigma^{MS})$. This function initializes $EWFN_{Linda}$ by assigning a multiset of tokens to each place. It is also allowed that the expression is missing, i.e. a place is initialized with the empty multiset.*
- $L_{write} : F_{tp} \rightarrow \Sigma$ *is the* Linda write function *(sometimes also called "tuple production rule") that determines the token to be written by each outgoing arc of a transition. Writing an empty tuple (ϵ) means that no tuple is written.*

Definition 9 (tuple element)**.** *A tuple element is a tuple $\langle p, tu \rangle$, $p \in P$, $tu \in \Sigma$. The set of all tuple elements $TE = P \times \Sigma$. We will use this structure for the definition of a* marking.

Definition 10 (marking)**.** *A* marking *M^{MS} is a multiset over the set of tuple elements TE. Each place may contain one or more equal tuples, thus the marking is defined as a multiset. Note that we may also use M^{MS} as a function returning the multiset of tuples on a given place, i.e. $M^{MS} : P \rightarrow \mathcal{P}(\Sigma^{MS})$ such that $\forall p \in P : M^{MS}(p) \in \mathcal{P}(\Sigma^{MS})$.*

Definition 11 (initial marking)**.** *The* initial marking *M_0^{MS} is created when applying the initialization function I_0 to all places, i.e. $\forall p \in P : M_0^{MS}(p) = I_0(p)$*

Definition 12 (template matching)**.** *$\approx$ is a binary relation over the sets Σ and*

[1] $\mathcal{P}(X)$ denotes the power set of a set X.

X, specifying if a template te matches a tuple tu: $\approx \subseteq \Sigma \times X$.

$$\langle tu, te \rangle \in \approx \text{ iff } |tu| = |te| \wedge \left(\forall n \in 1..|te| : \pi_n(te) = \pi_n(tu) \vee \pi_n(te) = \star\right)$$

$\pi_i(t)$ returns a projection to the ith element of a tuple tu, $|tu|$ denotes the *length* of a tuple, i.e. the number of elements it contains.

A template matches a tuple iff both have the same number of elements and each concrete value in the tuple equals the value on the same position in the template, or the template has a wildcard on this position.

Definition 13 (L_{read})**.** *Linda* `read` *operations (destructive and non-destructive) are formalized as a function $L_{read} : X \times \mathcal{P}(\Sigma^{MS}) \rightarrow \Sigma$. According to Linda's semantics [Gel85], only one tuple is returned regardless of the number of matching tuples. It is not determined which tuple of the set of matching tuples is returned.*

$$L_{read}(te, tu^{MS}) = \begin{cases} tu \in tu^{MS}, & \text{if } \exists tu \in tu^{MS} : tu \approx te \\ \epsilon, & \text{otherwise} \end{cases}$$

Note that the Linda matching function is independent of a concrete EWFN, it is thus not part of the definition of the EWFN tuple.

4.2.2 Semantics

The set of places that have arcs pointing to transition t is denoted as $\bullet t = \{p \mid pFt\}$[1], the set of transitions that have arcs pointing to place p is denoted as $\bullet p = \{t \mid tFp\}$, with F being the flow relation. $t\bullet$ and $p\bullet$ are defined accordingly.

A transition $t \in T$ that executes a *destructive* `read` operation (a.k.a. `take`) changes marking M_1^{MS} to M_2^{MS} as follows:

$$\forall p \in \bullet t : M_2^{MS}(p) = M_1^{MS}(p) - L_{\text{read}}(A(p, t, \text{“take”}), M_1^{MS}(p))$$

A transition $t \in T$ that executes a *non-destructive* `read` operation (a.k.a.

[1] The notation pFt is a shortcut denoting that $\langle p, t \rangle \in F$.

`read`) in contrast, does not have any effect on the marking:

$$\forall p \in \bullet t : M_2^{MS}(p) = M_1^{MS}(p)$$

Definition 14 (enabled)**.** *A transition $t \in T$ is called* enabled *in marking M^{MS} iff*

$$\forall p \in \bullet t : L_{read}(A(p,t), M^{MS}(p)) \neq \epsilon$$

Definition 15 (satisfied)**.** *A template $te \in X$ is called* satisfied *on multiset tu^{MS} iff $L_{read}(te, tu^{MS}) \neq \epsilon$. This can also be written as function $Sat : X \times \mathcal{P}(\Sigma^{MS}) \rightarrow \mathbb{B}$.*

$$Sat(te, tu^{MS}) = \begin{cases} \texttt{true}, & \text{if } L_{read}(te, tu^{MS}) \neq \epsilon \\ \texttt{false}, & \text{otherwise} \end{cases}$$

Definition 16 (fire)**.** *A transition $t \in T$ that is enabled in marking M_1^{MS}* may fire *and change marking M_1^{MS} to M_2^{MS} as follows:*

$$\begin{aligned} \forall p \in \bullet t : M_2^{MS}(p) &= M_1^{MS}(p) - L_{read}(A(p, t, \text{“take”}), M_1^{MS}(p)) \\ \forall p \in t\bullet : M_2^{MS}(p) &= M_1^{MS}(p) + L_{write}(t, p) \end{aligned}$$

Note that in this definition, the operators $+$ and $-$ are defined on multisets.

In the following, we denote the change from marking M_1^{MS} to M_2^{MS} when transition t_1 fires by the expression $M_1^{MS} \xrightarrow{t_1} M_2^{MS}$.

Definition 17 (firing sequence)**.** *There are two kinds of firing sequences: (i) a* finite firing sequence *is a sequence of markings*

$$M_1^{MS} \xrightarrow{t_1} M_2^{MS} \xrightarrow{t_2} \dots \xrightarrow{t_n - 1} M_n^{MS} \xrightarrow{t_n} M_{n+1}^{MS}$$

such that $n \in \mathbb{N}^*$ and $\forall i \in 1..n : M_i^{MS} \xrightarrow{t_i} M_{i+1}^{MS}$

M_1^{MS} is called start marking*, M_{n+1}^{MS} is called* end marking*, n is the* length *of the sequence. (ii) An infinite firing sequence is a sequence of markings*

$$M_1^{MS} \xrightarrow{t_1} M_2^{MS} \xrightarrow{t_2} \dots$$

such that $\forall i \in \mathbb{N}^* : M_i^{MS} \xrightarrow{t_i} M_{i+1}^{MS}$

Since EWFNs are Petri nets, the following definitions define standard properties known from this field for EWFNs as well.

Definition 18 (reachable)**.** *A marking M_j^{MS} is called* reachable *from a marking M_i^{MS} iff there is a finite firing sequence with M_i^{MS} as start marking and M_j^{MS} as end marking, i.e. $\exists n \in \mathbb{N} : M_i^{MS} \xrightarrow{t_1} \ldots \xrightarrow{t_n} M_j^{MS}$. M_j^{MS} is called* reachable *from M_i^{MS} in n steps.*

Definition 19 (live)**.** *$EWFN_{Linda}$ is called* live *iff starting from every marking reachable from the initial marking and for every transition $t \in T$ there is a marking M^{MS} where t is contained in the firing sequence.*

Definition 20 (bounded)**.** *$EWFN_{Linda}$ is called* bounded *iff for each place $p \in P$ there is an $n \in \mathbb{N}$ such that p contains at most n tokens in every reachable marking M^{MS}.*

Definition 21 (save)**.** *$EWFN_{Linda}$ is called* save *iff for each place $p \in P$, p contains at most 1 token in every reachable marking M^{MS}.*

4.2.3 Detection of Conflicts

It is important to note that the templates of tuplespace operations may overlap, i.e. if one of a set of different transitions destructively reads from the same place with templates that match the same tuple, a *conflict* is created. Or, more formally: a transition t is in conflict with transition t' in marking M^{MS} iff

$$\begin{aligned} &t, t' \text{ are enabled in } M^{MS} \wedge \\ &\exists p \in \bullet t \cap \bullet t' : L_{\text{read}}(A(p,t), M^{MS}(p)) = L_{\text{read}}(A(p,t'), M^{MS}(p)) \wedge \\ &\qquad\qquad (\langle p, t, \text{“take”}\rangle \in F \vee \langle p, t', \text{“take”}\rangle \in F)) \end{aligned}$$

According to Linda semantics [Gel85], this conflict is resolved non-deterministically.

In order to be able to statically detect conflicts (i.e. during design time), we define intersection and comparison operators on templates.

Definition 22 (template equality). *Two templates te_1 and te_2 are equal (denoted as $te_1 \doteq te_2$) iff for every marking $M^{MS'}$, both match the exact same set of tuples:*

$$\forall M^{MS'} \in \mathcal{P}(M^{MS}) : \{tu \mid tu = L_{read}(te_1, M^{MS'})\} = \{tu \mid tu = L_{read}(te_2, M^{MS'})\}$$

Definition 23 (template comparison). *Template te_1 is said to be "smaller or equal" to te_2 (denoted as $te_1 \dot{\subseteq} te_2$) iff for every marking $M^{MS'}$, te_1 describes a subset of the tuples that are described by template te_2: $\forall M^{MS'} \in \mathcal{P}(M^{MS}), A^{MS} = \{tu \mid tu = L_{read}(te_1, M^{MS'})\}, B^{MS} = \{tu \mid tu = L_{read}(te_2, M^{MS'})\}$:*

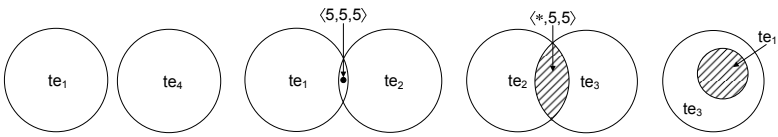
$$A^{MS} \leq B^{MS}$$

Definition 24 (template intersection). *The result of the intersection of the tuples described by two templates te_1 and te_2 can again be described as a template te_3, such that $\forall M^{MS'} \in \mathcal{P}(M^{MS}), A^{MS} = \{tu \mid tu = L_{read}(te_1, M^{MS'})\}, B^{MS} = \{tu \mid tu = L_{read}(te_2, M^{MS'})\}, C^{MS} = \{tu \mid tu = L_{read}(te_3, M^{MS'})\}$:*

$$C^{MS} = A^{MS} \cap B^{MS}$$

te_3 is the template that describes the set of tuples resulting from the expression $A^{MS} \cap B^{MS}$. We denote template intersection using the symbol $\dot{\cap}$, i.e. $te_3 = te_1 \dot{\cap} te_2$.

Note that an empty intersection is denoted by the empty tuple (ϵ).



(a) Disjoint Templates (b) Single Overlap (c) Multi Overlap (d) Containment

Figure 4.2: Template Intersection

In the following, we present an algorithm to calculate the intersection of a given set of templates. Consider the templates $te_1 = \langle 5, \star, 5 \rangle$, $te_2 = \langle \star, 5, \star \rangle$,

$te_3 = \langle \star, \star, 5 \rangle$, $te_4 = \langle 4, 3, 2 \rangle$ and their intersections depicted in Figure 4.2. Generally, four cases can be distinguished:

Disjoint templates The templates are disjoint and describe completely different sets of tuples, e.g. $te_1 \dot{\cap} te_4 = \epsilon$.

Single overlap Both templates describe sets of tuples that have only one single tuple in common, e.g. $te_1 \dot{\cap} te_2 = \langle 5, 5, 5 \rangle$.

Multi overlap The intersection of the sets of tuples described by the templates te_2 and te_3 results again in a set of tuples, which can be described by a template, e.g. $te_2 \dot{\cap} te_3 = \langle *, 5, 5 \rangle$

Containment A template describes a super-set of another, e.g. all tuples described by te_1 are also described by te_3, but te_3 describes even more tuples than te_1, i.e. $te_1 \dot{\subset} te_3$.

Equality is a special case of containment where both templates describe the exact same set of tuples, e.g. $te_2 \dot{=} te_2$.

Algorithm 4.1 describes how template intersection can be calculated. It takes a set of templates with equal length as input and calculates their intersection in form of a template. The general idea of the algorithm is described by Figure 4.3: The templates are layered on top of each other and a set (tmp) containing the element on position i of all templates is created (depicted by the box around column three in Figure 4.3) and checked for certain properties; if there are two different values in tmp that are both not wildcards ($\star$), the intersection is empty (see (1) in Algorithm 4.1 for the formal expression of this condition and column 1 in Figure 4.3 for an example). If all values in set tmp are equal, this value (that might either be a wildcard or an actual value) is added to the result template at position i (see (2) in Algorithm 4.1 for the formal expression of this condition and column 2 and 3 in Figure 4.3 for an example).

Finally, if both conditions before are not satisfied, the only possibility left is that tmp contains one or more equal actual values (or "actuals" for short) and one or more wildcard characters (see (3) in Algorithm 4.1 and the boxed column 4 in Figure 4.3 for an example)). In this case, the value of the actuals

Algorithm 4.1 Template Intersection

Require: Te, set of templates with equal length (equal number of elements) to be intersected. $\exists n : \forall\ \text{te} \in \text{Te} : |\text{te}| = n$

```
function INTERSECT(Te)
    res ← ϵ
    for i = 1 to |te|, te ∈ Te do
        tmp ← ∅
        for all te ∈ Te do
            tmp ← tmp ∪ π_i(te)
        end for
        if ∃x, y ∈ tmp : x ≠ y ∧ x ≠ ⋆ ∧ y ≠ ⋆ then          (1)
            return ϵ
        else if ∀x, y ∈ tmp : x = y then                      (2)
            π_i(res) ← x
        else                                                  (3)
            π_i(res) ← x, x ∈ tmp \ {⋆}
        end if
    end for
    return res
end function
```

is added to the resulting template describing the intersection. The precondition for the algorithm is that all templates to be intersected are of equal size, otherwise the empty tuple is returned immediately. The result of the intersection of

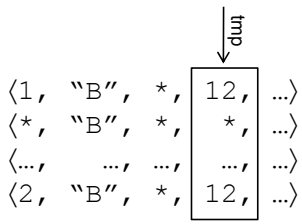


Figure 4.3: Illustration of the Template Intersection Algorithm

the templates shown in Figure 4.3 is ϵ, since there are two different actuals on the first position of template 1 and 4 (counting from the top).

Proposition 1 (Termination of Algorithm 4.1)**.** *Algorithm 4.1 terminates for any given input.*

Proof. Termination follows directly from the definition of the algorithm: the nested for-loops are either both executed entirely (i.e. until their exit condition evaluates to `true`) or the outer loop is exited prematurely (i.e. when the condition at (1) evaluates to `true`). □

Proposition 2 (Correctness of Algorithm 4.1)**.** *Algorithm 4.1 generates a (possibly empty) template that matches only tuples that also match* all *individual templates used to calculate the intersection template.*

More formally: Let Te be the set of templates to be intersected and template te' be the result of the intersection. For every marking M^{MS}, the tuples $G = \{tu \mid tu = L_{read}(te', M^{MS})\}$ matching the calculated (intersection) template te' also match all individual templates $te \in Te$, i.e.

$$\forall tu \in G \ \forall te \in Te : \langle te, tu \rangle \in \approx$$

Note that, since we are comparing sets, we interpret the appliance of a template in the following proof to return *all* matching tuples (such an operation will later be called `readall`). This stays in contrast to the classical Linda `read` or `take` operations which non-deterministically return only one tuple amongst the set of matching tuples, and thus are unsuitable for comparing sets of matching tuples.

Proof. We proof that the calculated template matches only tuples that also match all individual templates by looking at the properties of the row of actuals and wildcards called tmp in Algorithm 4.1 and Figure 4.3.

tmp contains different actuals In this case, there are different actuals on the same position of the templates. There is no tuple that matches all templates. Consequently, the intersection is the empty template, which

does not match any tuple. This case is implemented by position (1) in Algorithm 4.1.

tmp contains only equal elements (wildcards or actuals) Here, the resulting template has the actual or the wildcard on the position in question. In the case of the actual, all templates have the same actual on this position. The resulting template thus also needs to have this actual on the position. In the case of a wildcard, all templates have a wildcard on this position. The resulting template thus can safely use a wildcard on this position since tuples with any actual on this position will match all templates. This case is implemented by position (2) in Algorithm 4.1.

tmp contains a mix of equal actuals and wildcards The only possibility left is that tmp contains a mix of actuals and wildcards, where all actuals are equal. In this situation, the actual is written to the resulting template since only tuples with the actual on the current position match all templates. Position (3) implements this case in Algorithm 4.1.

□

We can now define the notion of a *conflict-free transition*, i.e. a transition that is not in conflict with any other transition for any given marking M^{MS}. Note that since we have provided an algorithm that calculates the intersection of templates which is only based on the elements of the templates itself (and is thus not depending on any marking), this check can be applied during design time, e.g. as part of a static validation of a given EWFN.

Definition 25 (conflict-free transition)**.** *A transition $t \in T$ is called* conflict-free *iff*

$$\forall p \in \bullet t \ \forall t' \in p \bullet \setminus \{t\} : A(p,t) \dot{\cap} A(p,t') = \epsilon \ \vee \\ \big(\langle p, t, \text{“read”}\rangle \in F \wedge \langle p, t', \text{“read”}\rangle \in F\big)$$

A transition t is conflict-free iff the intersection of templates of arcs from different transitions consuming from shared places with t is empty, or every

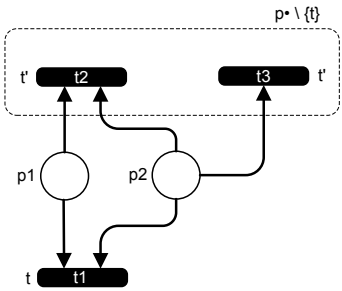


Figure 4.4: Example for the Calculation of t' in Definition 25

transition in question issues only non-destructive `read` operations. The motivation of the first requirement – an empty intersection of the templates from different transitions sharing a place – is depicted in Figure 4.4: transition t_1 is in potential conflict with transitions t_2 and t_3 over both places p1 and p2. Transition t_1 thus is only *conflict free* if the intersection of the templates on its arcs leaving p1 and p2 with all other templates on the arcs leaving the places is empty.

4.2.4 Join Matching

Figure 4.5 presents a transition that consumes tuples from two different places. The tuples in the places are structured according to Definition 8, i.e. contain two header fields denoting process model and instance ID. Consider tuple $1 = \langle 3, 7, \text{“}CF\text{”} \rangle$, tuple $2 = \langle 3, 6, \text{“}CF\text{”} \rangle$ and tuple $3 = \langle 1, 1, \text{“}CF\text{”} \rangle$. Although all input places of transition t_1 contain tuples, it should not be enabled since these tuples are either from different process models, process instances or both. In this example, transition t_1 should only be enabled if there is at least one tuple in each input place that contains equal process and instance IDs, e.g. if there is a tuple $4 = \langle 3, 6, \text{“}CF\text{”} \rangle$ present in the right-most place. Only in this case, there are tuples with equal process ID and instance ID (the tuples 2 and 4 in this example) available in both places. In order to express this restriction, we introduce *join-variables*, which are indicated by a name starting with “?” in the

template.

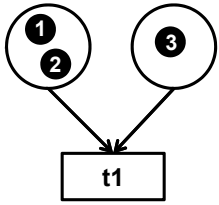


Figure 4.5: Example of a Transition Using Join-Matching

A join-variable is a special kind of wildcard that enforces equal values on its position across templates. Consider the templates te_1 and te_2 using join-variables for enforcing equal process ID and instance ID (called pid and iid respectively):

$$te_1 = \langle ?\text{pid}, ?\text{iid}, \text{“}CF\text{”} \rangle$$
$$te_2 = \langle ?\text{pid}, ?\text{iid}, \text{“}CF\text{”} \rangle$$

The transition using two separate `take` operations with te_1 and te_2 as templates is only enabled if there are tuples available in both incoming places that have equal values on their first and second position.

We extend the template matching from Definition 13 to be able to understand join-variables as fields in a template tuple. Note that for the matching itself a join-variable is treated as a wildcard ($\star$).

Definition 26 (join matching)**.** *A transition $t \in T$ that uses* join-variables *in its template operations is enabled in marking M^{MS} iff*

$$\forall p \in \bullet t : L_{read}(A(p,t)[?*/\star], M^{MS}(p)) \neq \epsilon \wedge$$
$$\forall p_1, p_2 \in \bullet t \ \ \forall n \in \mathbb{N}, \pi_n(A(p_1,t)) \textit{ is join-variable} :$$
$$\pi_n(A(p_2,t)) \textit{ is join-variable } \wedge \pi_n(A(p_1,t)) = \pi_n(A(p_2,t)) \wedge$$
$$\pi_n(L_{read}(A(p_1,t)[?*/\star], M^{MS}(p_1))) = \pi_n(L_{read}(A(p_2,t)[?*/\star], M^{MS}(p_2)))$$

Join-variables allow to express a restriction on the enablement rule of a transition such that it is enabled iff every template of its `read`/`take` operations

that use a join-variable is satisfied, and the tuple elements on the position of the join-variable are equal for each join-variable. The treatment of join-variables for the actual matching is expressed by the regular expression [?∗/⋆], meaning that every variable that starts with a ? is replaced by a wildcard (⋆).

4.2.5 Structural Properties

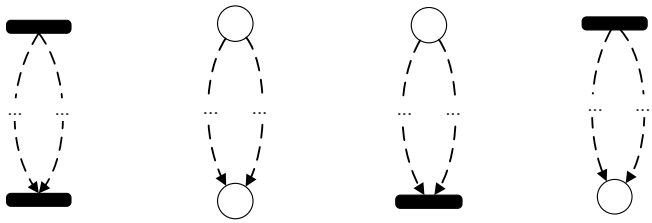
Similar to [vdA98], we define constraints on the structural level of $\text{EWFN}_{\text{Linda}}$ that allow us to detect design problems such as unbalanced splits and joins (e.g. an OR-split joined by an AND-join) with places and transitions. This is especially useful when directly using EWFNs as a modeling language, i.e. in cases where an $\text{EWFN}_{\text{Linda}}$ is not generated by transformation algorithms from higher-level programming languages but where it is modeled "by hand". A well-structured $\text{EWFN}_{\text{Linda}}$ must have a corresponding join transition to every fork transition, or a corresponding join place for every place that has more than one outgoing arc.

Definition 27 (well-structured)**.** *EWFN_{Linda} is* well-structured*, iff for any pair of nodes (a node is a place or a transition) $\langle x, y \rangle$ with $(x \in P \wedge y \in T) \vee (x \in T \wedge y \in P)$ which define the sets $S_1, S_2 \subseteq \mathcal{P}(P \cup T)$ that contain the nodes traveled through in the firing sequence from node x to node y, it holds that $(S_1 \cap S_2 \neq \{x, y\}) \vee (S_1 = S_2)$.*

Figure 4.6 shows examples of well-structured EWFNs. Well-structuredness requires forks and corresponding joins to be balanced, i.e. a fork modeled by a transition must be joined by a transition (see 4.6(a)), similarly, a split modeled by a place must be joined by a place (see 4.6(b)). Combinations of fork/join with transition and place (see 4.6(c) and 4.6(c)) are not allowed since such constructs need to be carefully designed in order not to produce a deadlock or other undesired behavior. In a fork/join example modeled as in Figure 4.6(d) the fork must ensure for any token that arrives that only one token is produced at the outgoings arcs, since the join is facilitated by a place that essentially implements merge semantics thus leading to duplicate tokens and ultimately lack of synchronization. The symmetric case (Figure 4.6(c)) is

similarly problematic since the join modeled by the transition can only receive one token in total (since the corresponding place above is a non-deterministic choice), however it expects one per incoming arc. A deadlock is the result.

It is important to note that the restriction of well-formedness does not imply a restriction on the expressiveness on the model itself: case 4.6(d) can be modeled as in 4.6(a) so that the join transition consumes all incoming tokens resulting in only one outgoing token. Similarly, case 4.6(c) can be modeled as in 4.6(b) in order to ensure that no deadlock is produced.



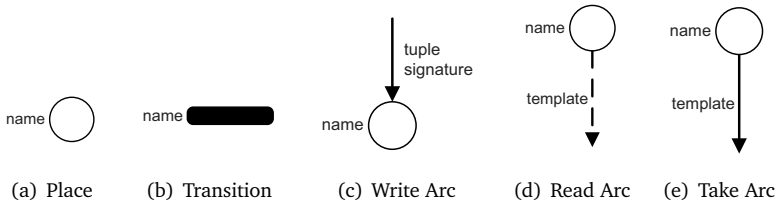
(a) Well-Structured (b) Well-Structured (c) Non Well-Structured (d) Non Well-Structured

Figure 4.6: Well-Structured and Non Well-Structured EWFNs

4.2.6 Graphical Notation

As with traditional Petri nets, EWFNs also have a graphical representation. The basic elements are depicted in Figure 4.7. The graphical notation again emphasizes on the close relationship between Petri nets and tuplespaces: each element can be mapped to an entity in a tuplespace-based application. The $\text{EWFN}_{\text{Linda}}$ notation therefore can also be used to graphically design tuplespace applications; the description of each element also mentions its counterpart in a tuplespace-based application.

Tuplespaces (Figure 4.7(a)) implement $\text{EWFN}_{\text{Linda}}$ places. Essentially, spaces act as a buffer for the tuples communicated in $\text{EWFN}_{\text{Linda}}$. Operations on tuplespaces – so called "coordination primitives" – are: `write` 4.7(c), `read` 4.7(d) and `take` 4.7(e) depicted by different kinds of arcs (see also set F in



(a) Place (b) Transition (c) Write Arc (d) Read Arc (e) Take Arc

Figure 4.7: $\text{EWFN}_{\text{Linda}}$ Graphical Elements

Definition 8). Note that write arcs are annotated with the signature of the tuple to be written, i.e. a tuple consisting of types (of set Φ) denoting the structure of the tuple to be written. Read arcs implement Linda semantics [Gel85], i.e. they block until their template (in tuple form, see Section 4.2.2) is *satisfied* (according to Definition 15).

Transitions (Figure 4.7(b)) are active elements in the network, i.e. they are the part where coordination primitives and ultimately program logic are executed. A transition is implemented as program that is client to one or more tuplespaces and actively retrieves and writes tuples to them. The "interface" of a transition is documented in $\text{EWFN}_{\text{Linda}}$ by (i) the template associated with each `read` or `take` arc (Figure 4.7(d) and 4.7(e)) pointing towards it and (ii) the tuple signature of each `write` arc that leaves this transition. Arcs in EWFNs therefore denote possible ways how tuples flow through the network.

4.2.7 Examples

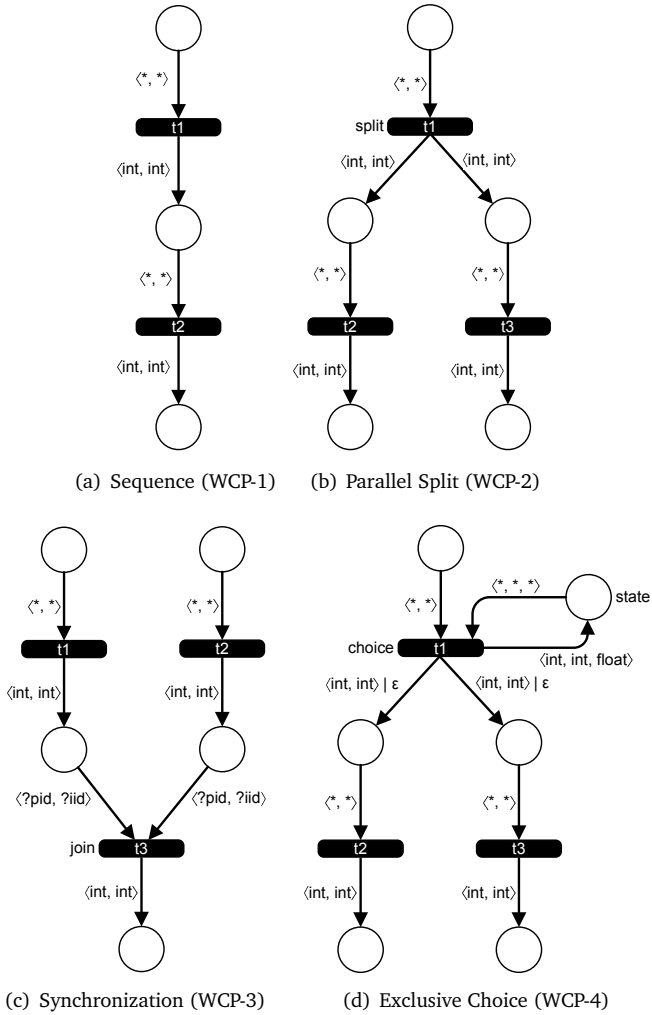
To emphasize on the applicability of $\text{EWFN}_{\text{Linda}}$ to workflow execution, we present the implementation of basic workflow control flow patterns that were initially documented in [vdAtHKB03] and revised in [RtHvdAM06]. Please note that in these examples, we consider the process instances implemented by the EWFNs to be *isolated*, i.e. all places contain tuples from only one process instance. This furthermore means that tuples from different process instances cannot arrive at the same transition.

In practice however, it is of course possible to have tuples from different process instances in the same application described by $\text{EWFN}_{\text{Linda}}$. Moreover, tuple matching in our tuplespace implementation (discussed in Chapter 7) is considering data types, e.g. the wildcard operator does not universally match all data in a specific field of a tuple, but only matches all data of a specific type[1]. The exact notation for a * consequently would be e.g. *:int for a * operator that matches all integer values, but not floating point numbers or strings.

Figure 4.8 shows the first four basic workflow control flow patterns modeled as $\text{EWFN}_{\text{Linda}}$. These patterns are similar to the control flow concepts initially proposed by the Workflow Management Coalition (WfMC) [Coa99].

A *sequence* describes activities that are enabled consecutively, one after another. In workflow languages, this is expressed by control flow edges between activities that do not have associated guards or conditions. In $\text{EWFN}_{\text{Linda}}$, this behavior is expressed by *write* and *take* arcs that pass on exactly one tuple that enables the succeeding transition. Figure 4.8(a) shows an example of two consecutive enabled transitions (t_1 and t_2) and the arcs and templates that pass on tuples with $\langle int, int \rangle$ signature to facilitate sequential behavior.

[1] Similar to the data type matching functionality in the original Linda definition.



(a) Sequence (WCP-1) (b) Parallel Split (WCP-2)

(c) Synchronization (WCP-3) (d) Exclusive Choice (WCP-4)

Figure 4.8: Sequence, Parallel Split, Synchronization and Exclusive Choice as $\text{EWFN}_{\text{Linda}}$

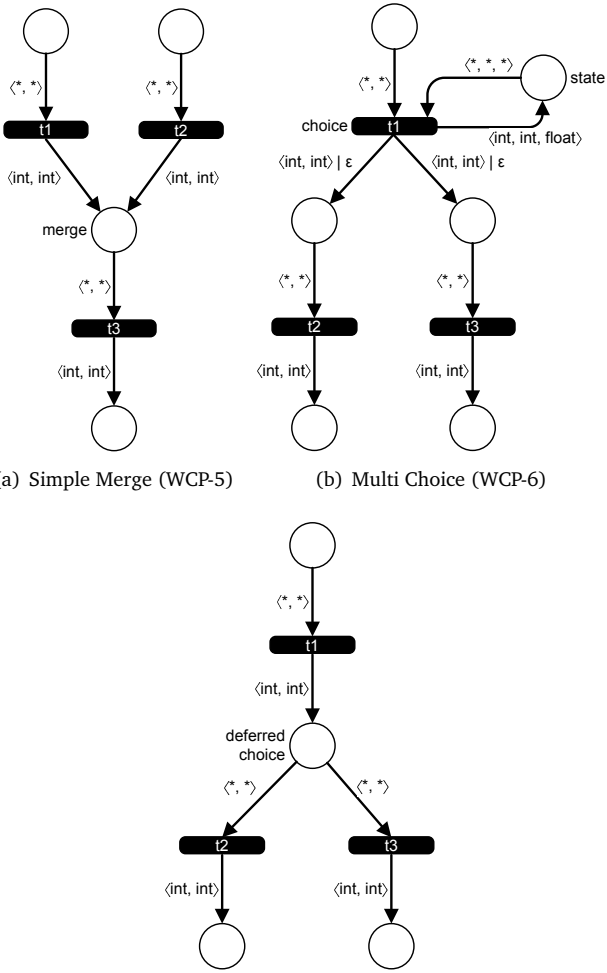
The *parallel split* pattern describes the divergence of a single thread of control flow into two or more parallel branches which are executed concurrently. In

$\text{EWFN}_{\text{Linda}}$, we express this behavior by a transition that is enabled by the arrival of a single tuple but then generates two tuples that each enable a succeeding transition. Figure 4.8(b) depicts this behavior: transition t_1, the "split" transition is enabled by a single incoming tuple matching $\langle *, * \rangle$, and writes two tuples $\langle int, int \rangle$ to consecutive places, enabling two transitions (t_2 and t_3) in parallel.

Synchronization describes the convergence of two or more threads of control flow into a single thread such that the subsequent branch is only enabled when all input branches have been enabled. In $\text{EWFN}_{\text{Linda}}$, this behavior is modeled using a transition that is only enabled if a tuple from each incoming branch is present. The transition then fires and writes a single tuple to its outgoing place in order to enable the succeeding transition. This behavior is depicted in Figure 4.8(c): the "join" transition t_3 is only enabled when a matching tuple is present in each of its input places. When enabled, transition t_3 writes a single tuple to its output place. Note that – although not necessary due to the preconditions described in the beginning of this section – join-variables (see Section 4.2.4) are used in the templates of transition t_3 to ensure that only tuples from the same process model and instance are considered. The use of join-variables (represented by ?iid and ?pid) in the templates of the incoming arcs of transition t_3 ensure that it is only enabled if the value in the first field of the tuple arriving at the left branch is equal to the value in the first field of the tuple arriving at the right branch. Similarly, the value in the second field (representing the process instance ID) of the arriving tuples must be equal. As a result, transition t_3 is only enabled if tuples with equal process model ID and process instance ID are available on its input places.

The *Exclusive choice* pattern describes the divergence of a single thread of control flow to precisely one of a number of outgoing branches. In workflow languages, this is used to express decisions that enable exactly one of a number of possible paths. In $\text{EWFN}_{\text{Linda}}$, this is modeled using a transition that is enabled upon reception of a tuple, and then writes exactly one tuple to one of its output places. All other places do not receive a tuple. Figure 4.8(d) depicts this behavior: upon enablement, transition t_1 writes either a tuple to the left or to the right outgoing place, depending on the state tuple stored in place "state".

Note that the ability to not write a tuple when firing is indicated by the ϵ tuple signature on the write arc as discussed in Section 4.2.



(a) Simple Merge (WCP-5)

(b) Multi Choice (WCP-6)

(c) Deferred Choice (WCP-16)

Figure 4.9: Simple Merge, Multi Choice and Deferred Choice as $\text{EWFN}_{\text{Linda}}$

Figure 4.9 depicts the next set of basic control flow patterns implemented as $\text{EWFN}_{\text{Linda}}$.

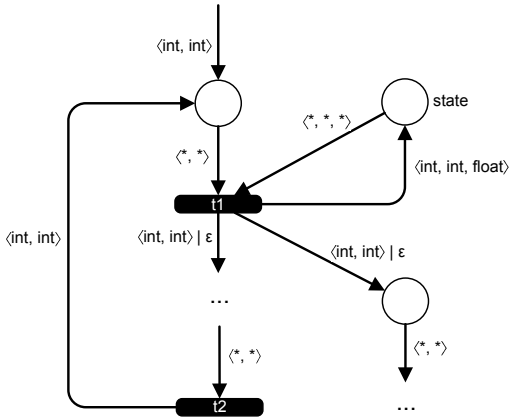
A *Simple Merge* describes the merge of two or more threads of control flow without synchronization, i.e. each enablement of an incoming branch results in control flow being passed to the subsequent branch. In $\text{EWFN}_{\text{Linda}}$, this is implemented using a "merge" place that merges incoming threads of control flow. Figure 4.9(a) shows an example of this behavior: place *merge* receives a tuple from either t_1 or t_2. Each time it receives a tuple, the subsequent transition (t_3) is enabled.

The *Multi Choice* pattern is a variation of the *Exclusive Choice* pattern, allowing *more than one* outgoing branch to be enabled. Since $\text{EWFN}_{\text{Linda}}$ focuses on the description of the interface (in terms of tuple structure) of the tuplespace application, there is no difference between the $\text{EWFN}_{\text{Linda}}$ pattern describing the *Exclusive Choice* and the *Multi Choice*: Internal behavior of the transition (i.e. the program consuming and producing tuples) determines whether to enable a single branch or multiple branches at once. Unlike in CP-Nets for instance where this would be expressed by CPN-ML expressions (i.e. little executable programs) on the incoming and outgoing arcs of a transition, this internal logic is not part of the $\text{EWFN}_{\text{Linda}}$ notation.

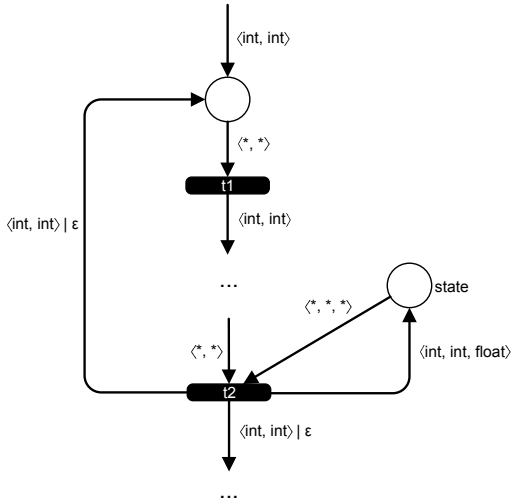
Deferred Choice describes the situation where two or more activities are enabled, but only one can be executed. After execution, all other – previously enabled – activities are disabled. This behavior is depicted in the $\text{EWFN}_{\text{Linda}}$ of Figure 4.9(c): a tuple in place "deferred choice" enables both transitions t_2 and t_3. Firing of either transition disables the other one by removing the tuple from its input place.

Figure 4.10 shows two structured loop patterns from the workflow control flow patterns and their implementation as $\text{EWFN}_{\text{Linda}}$. A structured loop is a construct that allows the repeated sequential execution of a set of activities and has exactly one entry and one exit point. *Pre-test* (commonly known as "while loop") and *post-test* (common known as "repeat ... until loop") loops are distinguished based on whether the test for the exit condition happens before or after execution of the loop body. When the exit condition evaluates to `true` in the repeat case or `false` in the while case, control flow is passed to the

immediate following activity after the loop body. In $\text{EWFN}_{\text{Linda}}$, a structured loop is implemented by passing a tuple from the last activity of the loop body back to the loop start and an exclusive choice (see Figure 4.8(d)) implementing the loop condition test at either the beginning or the end of the body of the loop. The exclusive choice is responsible to either start a new iteration of the loop or exiting it and passing control flow on to the immediately following activity. Figure 4.10(a) shows the $\text{EWFN}_{\text{Linda}}$ version of a "while loop", the exclusive choice evaluating the loop condition is placed before the body of the loop. In the repeat until loop in contrast (Figure 4.10(b)), the exit condition is evaluated after executing the loop body.



(a) While Loop



(b) Repeat-Until Loop

Figure 4.10: Structured Loops (WCP-16) Modeled as $\text{EWFN}_{\text{Linda}}$

4.3 Behavioral Equivalent Place-Transition Nets

In this section, we present a mapping from $\text{EWFN}_{\text{Linda}}$ into (standard) PT-nets. This allows to reuse the high number of existing PT-net analysis tools such as LoLa [Sch00] or PeP [Gra97]. Furthermore, it connects $\text{EWFN}_{\text{Linda}}$ to an established formalism so that it does not stand for itself, but is rather embedded into existing formalisms.

4.3.1 Place-Transition Nets

Before defining the actual mapping, we briefly summarize PT-nets by providing the most important definitions [Rei85].

Definition 28 (Place-Transition Net)**.** *A Place-Transition Net is a tuple* $PN = \langle P, T, F, W, I_0 \rangle$ *with:*

- P*, a finite, non-empty set of* places
- T*, a finite, non-empty set of* transitions *with* $P \cap T = \emptyset$
- $F \subseteq (P \times T) \cup (T \times P)$*, a set of arcs known as the* flow relation
- $W : F \to \mathbb{N}^*$*, the* weight-function *assigning a weight to each arc* $f \in F$*, denoting the number of tokens produced or consumed by a transition*
- $I_0 : P \to \mathbb{N}_0$*, the* initialization function *that assigns the initial number of tokens to a place, such that* $\forall p \in P : I_0(p) \in \mathbb{N}_0$

Definition 29 (marking)**.** *A marking* M *is a function* $P \to \mathbb{N}$*. The initial marking* M_0 *is the marking that equals the initialization function, i.e.* $\forall p \in P : M_0(p) = I_0(p)$

Please note that, for space reasons, we do not explicitly mention the definitions of *enabled*, *fire*, *firing sequence* and *reachable* as we do not need them for the definition of the mapping.

4.3.2 Mapping $\text{EWFN}_{\text{Linda}}$ to PT-Nets

The idea of the mapping to PT-nets is based on the same strategy that is used to map CP-nets into PT-nets [Jen92]: the path defined by the firing sequence of

each *different* tuple is represented by its own set of places and transitions. Since tokens in PT-nets do not have colors, different sets of places and transitions are used to separate the tokens from each other (i.e. the information carried by the tuple is added to the places and transitions through annotation). In that way, the initial, Colored Petri Net is unfolded to a behavioral equivalent PT-net.

As discussed, places and transitions are the vehicle used to separate tokens. Based on the $\text{EWFN}_{\text{Linda}}$ definitions from the previous sections, we start with the definitions of the respective sets necessary for the mapping and explain their construction.

Definition 30 (place elements). *The set of place elements PX is defined as* $PX = \{\langle p, x\rangle \mid p \in P, x \in \{y \in M^{MS}(p) \mid \langle y, z\rangle \in \approx, z = A(p,t), t \in p\bullet\}\}$

The set of place elements contains tuples of the form $\langle p, x\rangle$ for each tuple in $M^{MS}(p)$ that matches the template $A(p,t)$. This is the place "unfolding" as discussed before: each place is annotated with each different tuple that is read or taken from it.

Definition 31 (transition elements). *We define the set of transition elements TX as* $TX = \{\langle t, x\rangle \mid t \in T, x = \{\langle p, tu\rangle \mid p \in \bullet t, tu = L_{read}(A(p,t), M^{MS}(p)) :$ *t is enabled in* $M^{MS}\}\}$

Similarly, the set of transition elements contains tuples of the form $\langle t, x\rangle$ for each marking $M^{MS}(p), p \in \bullet t$ that enables transition t. This set "unfolds" the set of $\text{EWFN}_{\text{Linda}}$ transitions to separate transitions for each marking of the preceding places of transition t that enables it.

We can now construct PT-net*, a behaviorally equivalent PT-net of a given $\text{EWFN}_{\text{Linda}}$. Since both nets use partially equal set and function names, we add a "*" to each element when it is from PT-net*. Elements without a "*" in their name refer to $\text{EWFN}_{\text{Linda}}$.

Definition 32 (PT-net*). *A behaviorally equivalent PT-net called PT-net* with PT-net** $= \left\langle P^*, T^*, F^*, W^*, I_0^*\right\rangle$ *from a given* $EWFN_{Linda} = \left\langle \Sigma, \Phi, P, T, F, X, A, B, I_0, L_{write}\right\rangle$ *is defined by:*

1. $P^* = PX$

2. $T^* = TX$
3. $F^* = \{\langle\langle p,x\rangle,\langle t,y\rangle\rangle \in (P^* \times T^*) \mid L_{read}(A(p,t), M^{MS}(p)) = tu \wedge \langle p, tu\rangle \in y\} \cup$
   $\{\langle\langle t,y\rangle,\langle p,x\rangle\rangle \in (T^* \times P^*) \mid L_{write}(t,p) = tu \wedge \langle p, tu\rangle \in y\}$
4. $\forall \langle\langle p,x\rangle,\langle t,y\rangle\rangle \in F^* \cap (P^* \times T^*) : W^*(\langle\langle p,x\rangle,\langle t,y\rangle\rangle) = \left|L_{read}(A(p,t), M^{MS}(p))\right|$
   $\forall \langle\langle t,y\rangle,\langle p,x\rangle\rangle \in F^* \cap (T^* \times P^*) : W^*(\langle\langle t,y\rangle,\langle p,x\rangle\rangle) = \left|L_{write}(t,p)\right|$
5. $\forall \langle p,x\rangle \in P^* : I_0^*(\langle p,x\rangle) = \left|J^{MS}\right|$, $J^{MS} = \{tu | tu = x, tu \in I_0(p)\}$

The set of places P^* from PT-net* is equal to set PX; each original $\text{EWFN}_{\text{Linda}}$ place is unfolded into as many places as there are distinct tuples read from it. Similarly, the set of transitions T^* in PT-net* is equal to set TX, the set of transitions from the original $\text{EWFN}_{\text{Linda}}$ unfolded by the different markings $M^{MS}(p), p \in \bullet t$ that enable transition $t \in T$.

The flow relation F^* of PT-net* is defined depending on the flow of tuples through the $\text{EWFN}_{\text{Linda}}$, i.e. there is an arc between place $\langle p,x\rangle \in P^*$ and transition $\langle t,y\rangle \in T^*$ iff $L_{\text{read}}(A(p,t))$ returns a tuple that is equal to the tuple that was used to unfold the place, i.e. $L_{\text{read}}(A(p,t)) = x$ with x being one of the tuples used to unfold place $p \in P$. Similarly, there is an arc from transition $\langle t,y\rangle \in T^*$ to place $\langle p,x\rangle \in P^*$ iff $L_{\text{write}}(t,p) = x$, with x being the tuple used to unfold the target place $p \in P$. Furthermore, the tuple x is part of the set y, i.e. the marking of the places $\bullet t$ that enabled transition t.

The weight function W^* for each arc is calculated by the amount of tuples removed or added to a specific place.

The initialization function I_0^* for PT-net* is defined by calculating the amount of tuples of the initial marking that are equal to the tuple used to unfold place $p \in P$ of $\text{EWFN}_{\text{Linda}}$. The place $\langle p,x\rangle \in P^*$ of PT-net* contains as many tokens as there are tuples x present in $I_0(p)$.

$\text{EWFN}_{\text{Linda}}$ is similar to a CP-net, we therefore use the same technique that was used in [Jen92] to proof the behavioral equivalence of CP-nets and PT-nets to proof the behavioral equivalence of $\text{EWFN}_{\text{Linda}}$ and PT-nets. The proof shows that each $\text{EWFN}_{\text{Linda}}$ has exactly the same sets of markings and firing sequences

as the behavioral equivalent PT-net called PT-net*. In the following, we use the "*" notation introduced before: all elements with a "*" refer to PT-net*, while the ones without refer to $\text{EWFN}_{\text{Linda}}$.

Note: since multisets are functions $S \to \mathbb{N}$ (Definition 1), M^* can be expressed as a multiset over P^*. When we write M^{MS*}, we mean the function $M^* : P^* \to \mathbb{N}$.

Lemma 1. *Transition $\langle t, x \rangle \in T^*$ is enabled in M^{MS*} iff transition $t \in T$ is enabled in M^{MS}.*

Proof. This is a direct consequence of the transition unfolding facilitated by Definition 31. Set TX consists of a tuple $\langle t, x \rangle$ for each marking $x \subseteq M^{MS}$ of the directly preceding places of transition t that enables it. As a result, if transition $t \in T$ is enabled in marking M^{MS}, then there is a transition $\langle t, x \rangle \in T^*$ with $x \subseteq M^{MS}$ that is enabled in marking x. □

Theorem 1. *PT-net* is behaviorally equivalent to $\text{EWFN}_{\text{Linda}}$, since the following properties hold:*

1. $M^{MS} = M^{MS*}$
2. $M_0^{MS} = M_0^{MS*}$
3. $\forall M_1^{MS}, M_2^{MS} \in \mathcal{P}(M^{MS}) : M_1^{MS} \xrightarrow{t \in T} M_2^{MS} \Leftrightarrow M_1^{MS*} \xrightarrow{t^* \in T^*} M_2^{MS*}$

We show that the markings of both nets are equal (item 1 and 2) and that if there is a transition that leads from marking M_1^{MS} to M_2^{MS} in $\text{EWFN}_{\text{Linda}}$, then there is an equivalent transition in PT-net* leading from M_1^{MS*} to M_2^{MS*}.

Proof. We proof each item individually:

1. Following Definition 10, the marking M^{MS} is defined as a multiset over TE. By Definition 29, M^* is a function $P^* \to \mathbb{N}$. As mentioned before however, by Definition 1 the function M^* also represents a multiset over P^*. In the following, we thus use M^{MS*} to denote the marking of PT-net*. We show that the tokens in both markings are equal. The only difference in the markings is that in PT-net*, tokens may reside in different places than in the original EWFN due to place unfolding.

From Definition 32 (item 1) we have $P^* = \mathrm{PX}$. Moreover, by Definition 32 (item 3), there is an arc leaving from place $\langle p, x \rangle \in P^*$ if L_{read} reads tuple x from it. Similarly, there is an arc pointing towards place $\langle p, x \rangle \in P^*$ if L_{write} writes tuple x to place $p \in P$. As a result, each token written to or read from place $p \in P$ of the original $\text{EWFN}_{\text{Linda}}$ has a matching arc leaving or pointing towards place $\langle p, x \rangle \in P^*$ in PT-net*. Comparing the markings of both nets, their number of tokens does not change, and both markings represent the same state of the net. The difference is that in case of the EWFN, different tuples may reside in the same place while in the case of PT-net*, tuples are "anonymous" tokens that are separated by places. The type information of the tokens in PT-net* is not lost, it is encoded in the (name of the) places they reside in.

2. The initial marking of a PT-net is defined as $\forall p \in P^* : M_0^{MS*}(p) = I_0^*(p)$ (Definition 29). The initial marking of $\text{EWFN}_{\text{Linda}}$ is defined as $\forall p \in P : M_0^{MS}(p) = I_0(p)$ (Definition 11). Following Definition 32, item 5, we see that the initial marking of a place $\langle p, x \rangle \in P^*$ of PT-net* is the number of tuples that are equal to tuple $x = \pi_2(\langle p, x \rangle), \langle p, x \rangle \in P^*$. All tuples $I_0(p)$ assigned to a certain place $p \in P$ in the original $\text{EWFN}_{\text{Linda}}$ are distributed to the places $\langle p, x \rangle \in P^*$ based on the place annotation $x = \pi_2(\langle p, x \rangle)$. There is no tuple lost if every distinct tuple is part of the place annotation. By Definition 30, this is the case as each member of the marking $M^{MS}(p)$ is used to create the places $\langle p, x \rangle$ of PT-net*. As a result, the number of tuples does not change. Equal tuples reside in the same place, different tuples reside on different places in the generated PT-net*. The sum of tuples of the initial marking in PT-net* is equal to the sum of tuples of the initial marking of $\text{EWFN}_{\text{Linda}}$.
3. According to Lemma 1, a transition $\langle t, x \rangle \in T^*$ is enabled in M^{MS*} iff a transition $t \in T$ is enabled in M^{MS}. If transition $t \in T$ fires, it changes marking M_1^{MS} to M_2^{MS}; firing the corresponding transition $\langle t, x \rangle \in T^*, t \in T$ changes the marking in PT-net* from M_1^{MS*} to M_2^{MS*}.
   More specifically, according to Definition 16, firing transition $t \in T$ removes the following place elements $\langle p, x \rangle \in M^{MS}$ from marking M^{MS}:

$\{\langle p, x\rangle \mid x = L_{\text{read}}(A(p,t), M^{MS}(p)), p \in \bullet t\}$ which has the same effect on marking M^{MS*} as if transition $\langle t, x\rangle \in T^*$ fires. By Definition 31, the set of place elements $\langle p, \text{tu}\rangle = \pi_2(\langle t, x\rangle), \langle t, x\rangle \in TX$ (i.e. the marking) that enables transition $\langle t, x\rangle \in T^*$ (and is therefore used for unfolding the transitions in the original $\text{EWFN}_{\text{Linda}}$) is defined as $\{\langle p, \text{tu}\rangle \mid p \in \bullet t, \text{tu} = L_{\text{read}}(A(p,t), M^{MS}(p)) : \text{t is enabled in } M^{MS}\}$ which is equal to the set of place elements removed when firing transition $t \in T$.
The tuples produced when firing transition $t \in T$ are equal to the ones produced by transition $\langle t, x\rangle \in T^*$ since according to Definition 32 item 3, there is an arc leaving from transition $\langle t, x\rangle \in T^*$ for each tuple produced by $L_{\text{write}}(t, p), p \in t\bullet$.

□

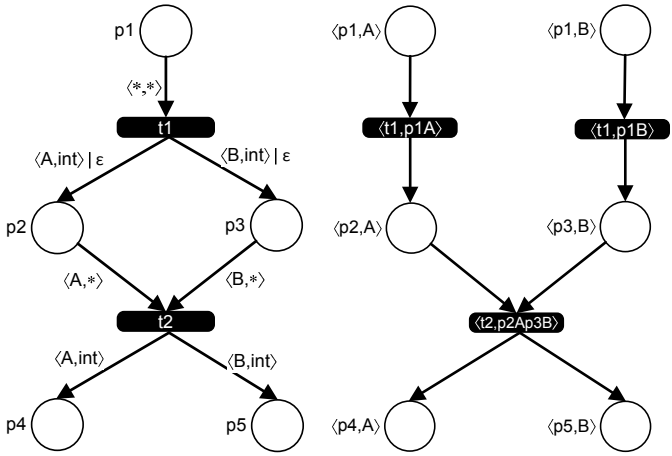
4.4 Example

In this section, we show an example of an $\text{EWFN}_{\text{Linda}}$ and its equivalent PT-net in both, graphical and tuple form. Figure 4.11(a) shows the $\text{EWFN}_{\text{Linda}}$, the corresponding PT-net is shown in Figure 4.11(b). The example EWFN depicts three interesting situations that occur when being unfolded: (i) the `take` arc between p_1 and t_1 is using wildcards in its template and matches more than one tuple in M^{MS} of $\text{EWFN}_{\text{Linda}}$. It is thus unfolded to multiple arcs. (ii) The fork on transition t_1 is unfolded to a sequence since transition t_1 is unfolded to one transition per tuple. (iii) The synchronizing join on transition t_2 is not unfolded since the synchronization of both paths must also take place in the PT-net to maintain the original semantics of $\text{EWFN}_{\text{Linda}}$.

We now define $\text{EWFN}_{\text{Linda}}$ in tuple form, reflecting the EWFN shown in Figure 4.11(a).

$\text{EWFN}_{\text{Linda}} = \langle \Sigma, \Phi, P, T, F, X, A, B, I_0, L_{\text{write}}\rangle$ with

- $\Sigma = (\text{“}A\text{“} \times \mathbb{N}) \cup (\text{“}B\text{“} \times \mathbb{N})$
- $\Phi = \{A, B, \text{int}\}$
- $P = \{p_1, p_2, p_3, p_4, p_5\}$
- $T = \{t_1, t_2\}$



(a) Example $\text{EWFN}_{\text{Linda}}$ Model (b) Semantically Equivalent PT-Net

Figure 4.11: Example for Mapping an EWFN into a PT-net

- $F = \{\langle p_1, t_1, \text{“}take\text{“}\rangle, \langle t_1, p_2\rangle, \langle t_1, p_3\rangle, \langle p_2, t_2, \text{“}take\text{“}\rangle, \langle p_3, t_2, \text{“}take\text{“}\rangle, \langle t_2, p_4\rangle, \langle t_2, p_5\rangle\}$
- $X = \{\langle *, *\rangle, \langle A, *\rangle, \langle B, *\rangle\}$
- $A(f) = \begin{cases} \langle *, *\rangle, & \text{if } f = \langle p_1, t_1\rangle \\ \langle A, *\rangle, & \text{if } f = \langle p_2, t_2\rangle \\ \langle B, *\rangle, & \text{if } f = \langle p_3, t_2\rangle \\ \epsilon, & \text{otherwise} \end{cases}$
- $B(f) = \begin{cases} \{\langle A, \text{int}\rangle, \epsilon\}, & \text{if } f = \langle t1, p_2\rangle \\ \{\langle B, \text{int}\rangle, \epsilon\}, & \text{if } f = \langle t1, p_3\rangle \\ \{\langle A, \text{int}\rangle\}, & \text{if } f = \langle t2, p_4\rangle \\ \{\langle B, \text{int}\rangle\}, & \text{if } f = \langle t2, p_5\rangle \\ \emptyset, & \text{otherwise} \end{cases}$
- $I_0(p) = \begin{cases} \{\langle A, 12\rangle, \langle B, 33\rangle\}, & \text{if } p = p_1 \\ \emptyset, & \text{otherwise} \end{cases}$

- $L_{\text{write}}(f) = \begin{cases} \langle A, 12\rangle, & \text{if } f = \langle t_1, p_2\rangle \\ \langle B, 33\rangle, & \text{if } f = \langle t_1, p_3\rangle \\ \langle A, 12\rangle, & \text{if } f = \langle t_2, p_4\rangle \\ \langle B, 33\rangle, & \text{if } f = \langle t_2, p_5\rangle \\ \epsilon, & \text{otherwise} \end{cases}$

The semantically equivalent PT-net is created by applying the construction rules in Definition 32. For better readability, we replace the sets of tuples that are part of the names of the created places with variables. Variable $A = \langle A, 12\rangle$, and $B = \langle B, 33\rangle$. Similarly, we use the variables $p_1A = \{\langle p_1, A\rangle\}$, $p_1B = \{\langle p_1, B\rangle\}$, $p_2Ap_3B = \{\langle p_2, A\rangle, \langle p_3, B\rangle\}$ to denote the markings that are part of the transition names in PT-net*.

PT-net* $= \langle P, T, F, W, I_0\rangle$ with

- $P^* = \{\langle p_1, A\rangle, \langle p_1, B\rangle, \langle p_2, A\rangle, \langle p_3, B\rangle, \langle p_4, A\rangle, \langle p_5, B\rangle\}$
- $T^* = \{\langle t_1, p_1A\rangle, \langle t_1, p_1B\rangle, \langle t_2, p_2Ap_3B\rangle\}$
- $F^* = \{\langle\langle p_1, A\rangle, \langle t_1, p_1A\rangle\rangle, \langle\langle p_1, B\rangle, \langle t_1, p_1B\rangle\rangle, \langle\langle t_1, p_1A\rangle, \langle p_2, A\rangle\rangle, \langle\langle t_1, p_1B\rangle, \langle p_3, B\rangle\rangle, \langle\langle p_2, A\rangle, \langle t_2, p_2Ap_3B\rangle\rangle, \langle\langle p_3, B\rangle, \langle t_2, p_2Ap_3B\rangle\rangle, \langle\langle t_2, p_2Ap_3B\rangle, \langle p_4, A\rangle\rangle, \langle\langle t_2, p_2Ap_3B\rangle, \langle p_5, B\rangle\rangle\}$
- $W^*(a) = 1$
- $I_0^*(p) = \begin{cases} 1, & \text{if } p = \langle p_1, A\rangle \\ 1, & \text{if } p = \langle p_1, B\rangle \\ 0, & \text{otherwise} \end{cases}$

4.5 Common Extensions

There are numerous extensions to Linda e.g. to solve the "multiple read problem" [RW96], a dedicated `update` operation [AS91] or multiset versions of the traditional Linda `read` and `write` operations [LMW99]. Although it is proven that the original Linda interface is turing-complete [BGZ97], these extensions increase productivity in practice. In this section, we therefore adopt four of the most prominent extensions – three additional operations and one extension to the graphical notation – and discuss how they fit into the formal

model described previously. Details and algorithms on how these tuplespace operations are implemented are given in Chapter 6.

Definition 33 (EWFN$_{\text{Extended}}$). *EWFN$_{Extended}$ is a tuple*

$$EWFN_{Extended} = \langle \Sigma, \Phi, P, T, S, F, X, A, B, I_0, L_{write} \rangle$$

EWFN$_{Extended}$ is based on EWFN$_{Linda}$, the sets Σ, Φ, P, T, X and the functions A, B, I_0 and L_{write} remain unchanged. The differences to EWFN$_{Linda}$ are:

- *Set S denotes the set of* arcs*, replacing the flow relation F from EWFN$_{Linda}$.*
- *Set F is now called the* node function $F : S \rightarrow (T \times P) \cup (P \times T \times R)$ *that assigns arcs from set S to tuples of the form $\langle$src, dest$\rangle$, the first element denoting the source node and the second the target node of the arc. The tuple is subject to the constraint that it must always contain a place and a transition, i.e. an arc is not allowed to be between two places or two transitions.*
- *Since most extensions introduce new tuplespace operations, set R is modified to reflect the new operations supported. In EWFN$_{Extended}$, $R = \{read, take, readall, takeall, update\}$*

In the following, we discuss each of these extensions individually.

4.5.1 Readall

The `readall` operation was initially proposed in [RW96], to solve the "Linda multiple read problem", i.e. the fact that executing a Linda `read` operation two times in a row with the same template may lead to the reception of different tuples. Recall that if there are multiple tuples that match a template in a `read` operation, the tuple that is returned is chosen non-deterministically by the tuplespace implementation. It is therefore not (easily) possible to retrieve all matching tuples for a given template – the same tuple may be returned multiple times.

In their work, Rowstron et al. [RW96] thus proposed the `copy-collect` primitive that creates a new space and copies all matching tuples into this

new location for further processing. Our version of `readall` is similar to the `copy-collect` primitive, but without creating a new space and copying the results over to that space. Instead, the set of matching tuples are returned directly to the client. Note that XVSM offers a similar solution to this problem in form of the `cnt(n)` selector [CKS09].

There are two places in the existing $\text{EWFN}_{\text{Linda}}$ formalism that need to be extended in order to support the new operation: (i) tuplespace operations are represented by arcs in the $\text{EWFN}_{\text{Linda}}$ graph, the `readall` operation thus introduces a new arc name that consequently needs to be added to the existing set R in Definition 8 as $R \cup \{\text{readall}\}$. (ii) Additionally, the new operation introduces a new matching function:

Definition 34 (L_{readall})**.** *Similarly to L_{read}, the `readall` operation is formalized as a function $L_{readall} : X \times \mathcal{P}(\Sigma^{MS}) \rightarrow \mathcal{P}(\Sigma^{MS})$. It returns all matching tuples for a given template te.*

$$L_{readall}(te, tu^{MS}) = \left\{ tu \mid tu \approx te, tu \in tu^{MS} \right\}$$

`readall` does not change the marking of a place. Upon firing of a transition $t \in T$ with a `readall` arc leaving from place $p \in P$ (i.e. $\langle p, t, \text{“}readall'' \rangle \in F$), the marking $M_2^{MS}(p)$ is equal to the marking $M_1^{MS}(p)$ before the transition fired.

$$M_2^{MS}(p) = M_1^{MS}(p)$$

An example for the use of the `readall` operation is presented in the patterns described in Chapter 5, e.g. in Section 5.2.1 that discusses the formalization of the BPEL `receive` activity.

4.5.2 Takeall

In contrast to the `readall` extension, `takeall` destructively removes all matching tuples of a given template and returns the matches to the client. A similar operation has been originally proposed as `collect` in [BWA94]. Instead of returning all matches to the client, `collect` creates a new space and leaves the result there for further processing.

Similar to `readall`, we extend the formal notation at two places to introduce `takeall` in our formalism. (i) We introduce a new arc name that is added to the existing set R in Definition 8: $R \cup \{\text{takeall}\}$. (ii) The new operation is formalized by the definition of the new matching function and a formal representation of its effect on the marking of a place p.

Definition 35 (L_{takeall})**.** *The* `takeall` *operation is formalized as a function* $L_{takeall} : X \times \mathcal{P}(\Sigma^{MS}) \rightarrow \mathcal{P}(\Sigma^{MS})$*. It returns all matching tuples for a given template* te*.*

$$L_{takeall}(te, tu^{MS}) = \left\{tu \mid tu \approx te, tu \in tu^{MS}\right\}$$

The marking $M_1^{MS}(p)$ of place $p \in P$ is changed to $M_2^{MS}(p)$ by a transition $t \in T$ that issues a `takeall` operation on place p (with $\langle p, t, \text{“}takeall''\rangle \in F$) as follows:

$$M_2^{MS}(p) = M_1^{MS}(p) - L_{\text{takeall}}(\text{te}, M^{MS}(p))$$

An example for the use of `takeall` is presented in the patterns described in Chapter 5, e.g. in Section 5.2.9 that discusses how to implement the BPEL `exit` activity.

4.5.3 Update

In traditional Linda, it is not possible to update fields in a tuple. Instead, the tuple must be read destructively (`take`), modified, and written back to the space. While the essential functionality is already present in the original Linda interface by using a sequence of `take` and `write` operations, it is much more convenient to cover the common case of updating a tuple with a single method.

Persistent Linda [AS91] recognized this need and consequently introduced the `in-out` operation that combines the operations `in` (equal to `take` in our system) and `out` (equal to `write` in our system) operation to a single operation to provide the functionality to update a tuple. Similarly, TSC [FKS+07] also directly supports update as an operation of their coordination interface.

In our formalization, we represent the `update` operation as another matching function L_{update}, and present two extensions to the existing formalism. (i) We introduce a new arc name that is added to the existing set R in Definition 8:

$R \cup \{\text{update}\}$. (ii) The new operation is formalized by the definition of the new matching function and a formal representation of its effect on the marking $M^{MS}(p)$ of a place $p \in P$.

Definition 36 (L_{update})**.** *The* `update` *operation is formalized as a function* $L_{update} : X \times \Sigma \times \mathcal{P}(\Sigma^{MS}) \rightarrow \Sigma$*. It replaces the chosen tuple matching* $te \in X$ *with* $tu \in \Sigma$ *and returns the old tuple, i.e. the tuple that was replaced by the new one. The matching is applied to the multiset of tuples* $tp^{MS} \in \mathcal{P}(\Sigma^{MS})$*.*

$$L_{update}(te, tu, tp^{MS}) = L_{read}(te, tp^{MS})$$

The marking $M_1^{MS}(p)$ of place $p \in P$ is changed to $M_2^{MS}(p)$ by a transition $t \in T$ that issues an `update` operation on place p (with $\langle p, t, \text{“}update'' \rangle \in F$) as follows:

$$M_2^{MS}(p) = M_1^{MS}(p) - L_{\text{update}}(\text{te}, tu, M^{MS}(p)) + \text{tu}$$

$\text{tu} \in \Sigma$ is the tuple that replaces the one matching $te \in X$.

An example for the use of `update` is presented in the patterns described in Chapter 5, e.g. in Section 5.2.1 that discusses the formalization of the BPEL `receive` activity.

4.5.4 Multiple Arcs Between Nodes

Some Petri net “dialects” such as CP-nets allow multiple arcs of the same direction between a single place and transition. This is for convenience reasons: if a transition in a CP-net consumes multiple tokens of different color from the same place, it is much more convenient (and the net is easier to understand) if there is a different arc for each color instead of having one arc with a complex expression consuming multiple tokens.

The argument to support multiple arcs of the same direction between the same pair of different nodes (place or transition) in EWFNs is similar: When modeling tuplespace-based applications, clients often consume more than one tuple from the same place. A transition for instance may consume a control

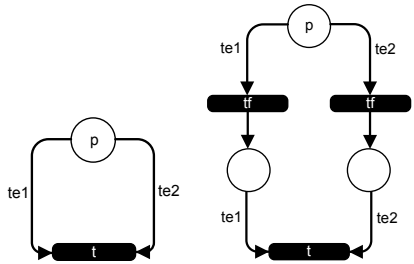
flow tuple as well as several data tuples. It is thus very convenient to be able to express this by drawing multiple arcs, each of them consuming one type of tuple.

To support multiple arcs as described before, we extend the formal notation of $\text{EWFN}_{\text{Linda}}$ in a way similar to how CP-nets support multiple arcs [Jen92]: (i) We remove the *flow relation* F and introduce S, a finite set of arcs such that $P \cap T = P \cap S = T \cap S = \emptyset$. (ii) We then re-introduce F as a function $F : S \rightarrow (T \times P) \cup (P \times T \times R)$, with $R = \{\text{read}, \text{take}, \text{readall}, \text{takeall}, \text{update}\}$ now called the *node function* that assigns arcs from S to tuples where the first field is the source, the second the destination node. These tuples are subject to the constraint that no arc may connect two places or two transitions, i.e. it is not allowed that the first and the second field of the tuples are of the same kind.

It is important to note however that we introduce this extension solely for reasons of convenience, no new semantics is introduced. In order to prove this claim, we show how to "simulate" EWFNs with multiple arcs using the $\text{EWFN}_{\text{Linda}}$ notation that does not include this extension. In the following, we present two different use cases – multiple arcs between a place and a transition and multiple arcs between a transition and a place – and show for each case that there is an equivalent $\text{EWFN}_{\text{Linda}}$.

Figure 4.12(a) shows a simple extended EWFN with two arcs between place p and transition t. Each arc is annotated with a different template, i.e. transition t reads two different tuples from place p and is thus only enabled iff both templates are satisfied. Figure 4.12(b) depicts a semantically equivalent EWFN, but without using multiple arcs: Here, "forwarding transitions" (t_f) are used to execute the templates individually – each transition t_f fires independent from the other – but both paths are synchronized again by transition t as it is done directly by transition t in Figure 4.12(a). As a result, the semantics of both EWFNs is identical. The drawback of the non-extended EWFN version is that one additional place and transition per arc in the multiple arc version is needed to implement the desired semantics.

In Figure 4.13 an equivalent situation with a transition and a place is shown. Figure 4.13(a) shows the extended EWFN with two arcs between transition

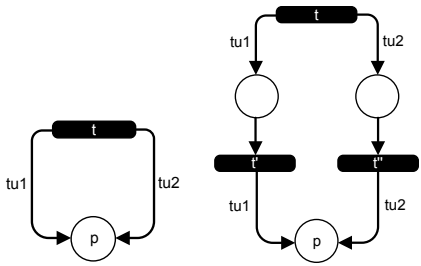


(a) Multiple Arcs Between a Place and Transition

(b) Equivalent Version with only one Arc Between a Place and a Transition

Figure 4.12: Multiple Arcs Between a Place and a Transition and an Equivalent Version with Single Arcs

t and place p. Upon firing, transition t writes two tuples tu_1 and tu_2 into place p. Figure 4.13(b) shows an equivalent EWFN without using multiple arcs between t and p. Instead, transition t writes both tuples tu_1 and tu_2



(a) Multiple Arcs Between a Transition and Place

(b) Equivalent Version with only one Arc Between a Transition and a Place

Figure 4.13: Multiple Arcs Between a Transition and a Place and an Equivalent Version with Single Arcs

into different, intermediary places. Subsequently, transitions t' and t'' fire and forward the tuples to their destination place p.

4.6 Conclusion

In this chapter, we introduced Executable Workflow Nets (EWFN), a Petri net dialect that provides a formal model for tuplespace-based applications. The most important aspect of this model is the absence of a central point of control – each transition (or tuplespace client) is decoupled from the remainder of the network. We provided a formal description of the syntax and semantics of the model, and then focused on advanced aspects like conflict detection, join matching and a behavioral equivalent mapping of $\text{EWFN}_{\text{Linda}}$ to standard Petri nets. Furthermore, a graphical notation was presented and the applicability to workflow execution through the implementation of basic workflow control flow patterns with EWFNs was demonstrated. It is important to note that EWFNs are a generic, low level execution language. We believe that most higher-level workflow definition language such as WS-BPEL or BPMN can be expressed in EWFNs.

Finally, we demonstrated in this chapter how common extensions to the Linda coordination interface, including `readall`, `takeall`, `update` and multiple arcs between places and transitions, can be integrated into EWFNs.

As the execution model for the decentralized workflow engine, EWFNs are the basis for the work presented in all chapters to follow. In the next chapter, we formalize WS-BPEL 2.0 with EWFNs, i.e. express the semantics of each activity in form of an EWFN pattern with the goal of decentralized execution of BPEL processes. In Chapter 6, we give details how EWFNs can be executed using tuplespaces. Chapter 7 then presents the architecture and implementation of a tuplespace middleware that is designed to efficiently execute EWFNs.

CHAPTER 5

TRANSFORMATION OF WS-BPEL 2.0 INTO EXECUTABLE WORKFLOW NETS

This chapter gives a detailed description of how executable BPEL 2.0 processes are transformed into EWFNs for the purpose of their decentralized execution. The most significant difference of our EWFN representation of WS-BPEL in comparison to related approaches is that we specifically focus on *execution*, i.e. all necessary elements to reproduce the original BPEL semantics are included. This means in particular that (i) our models include data flow, (ii) the communication with external Web services and (iii) explicit decisions. Related approaches such as [HSS05] or [OVvdA+05a] specifically focus on formal verification, they thus abstract from details such as data handling or internal behavior of an activity in favor of a small amount of places and transitions. Moreover, reduction rules are applied to further minimize the size of the created Petri net in order to reduce the state space to be traversed during model checking.

The EWFNs presented in this chapter are directly used for process execution i.e. they are the internal execution format of the decentralized workflow engine. To be able to represent BPEL in form of EWFNs, we create a composable pattern for each activity in the BPEL specification. The created EWFN patterns follow the same composition rules as their BPEL counterparts; the `scope` pattern for instance has exactly one primary activity whereas the `flow` or `sequence` patterns may contain a number of activities.

The description of each pattern is focused on how the semantics of a BPEL activity is implemented. Its semantics is summarized briefly and explained as needed. When the behavior of the activity depends on the state of the process instance, we play the "token game", i.e. we explain the different states of the EWFN and explain how the pattern implements the transition between them.

Before actually discussing EWFN patterns, Section 5.1 introduces BPEL specific tuple types and structures together with the graphical notation we use to document the patterns. Subsequently, we present EWFNs for basic activities in Section 5.2 and structured activities in Section 5.3. The `scope` activity and its associated handlers is presented in Section 5.4. An example in Section 5.5 showing a simple WS-BPEL process and its EWFN equivalent concludes this chapter.

In the following, we use the words "tuple" and "token" interchangeably. Also, we mean the same when we talk about a "contained" or an "inner" activity. To ease readability and to disambiguate the meaning of words from regular text and BPEL or EWFN constructs, BPEL activities are printed in `teletype` font and EWFN place names are typeset in *italic* letters.

5.1 Approach

For the transformation, we use an approach that is based on previous transformations of BPEL to Petri nets, specifically [Loh08], [Sta05] and [Pop08]. We follow their idea of representing each activity in the BPEL specification as a separate *pattern* that uses *interface places* in order to be composed with other patterns. We distinguish between two general patterns, corresponding to *basic* and *structured* activities in the WS-BPEL specification. Basic activities represent

elementary steps of the process, i.e. invoke a Web service, assign values to variables, immediately end the process instance, etc. Structured activities in contrast encode the control flow logic of the process and may contain other basic or structured activities.

A very important property of the resulting EWFN is that both, data and control flow are expressed explicitly; the BPEL way of handling data with scoped variables is transformed to explicit data flow arcs. These explicit data arcs are a prerequisite for the partitioning algorithms presented in [Wut10] that will be applied to the resulting EWFN.

5.1.1 EWFN Extensions

Before we provide details about the mapping, we introduce two extensions to the generic EWFN model: (i) the `sync` operation provides support for a *dynamic* number of incoming arcs, i.e. for cases where the precondition of a transition (i.e. the number of incoming arcs and their templates) must be defined at runtime. (ii) A BPEL specific extension to support different kinds of control flow, namely positive control flow and *dead-path*. Note that this is the only BPEL specific extension required. To implement it, we rely on the well-known technique of passing different kinds of control flow tokens to model Dead-Path-Elimination [LR00] in Petri net based systems. The basic idea for using token colors to propagate dead paths was first introduced in [LSW97] in the context of expressing EPK semantics in Petri nets, but was e.g. also used in [MvDvdA07] to provide a formal semantics for the infamous EPK OR-Join.

5.1.1.1 `sync`

Similar to the extensions presented in Section 4.5, the `sync` operation does not introduce new semantics to the EWFN model. It is basically the dynamic version of the extension that allows multiple arcs between the same place and transition (cf. Section 4.5.4). Instead of using multiple arcs and one template for each arc, only one arc and multiple templates are used. The important property of `sync` is that the number of templates does not need to be known at design time. Through defining the templates at runtime instead, `sync` allows

the pre-condition of a transition to be defined at runtime e.g. when the number of incoming arcs (and thus tuples to be read) is not known beforehand. The graphical representation of the `sync` arc is presented in Section 5.1.1.3.

The `sync` operation will also play an important role in Chapter 6 since it cannot be efficiently implemented using the coordination interface of current tuplespace middleware. Consequently, we present an algorithm that addresses this shortcoming.

5.1.1.2 Tuple Structure and Types

As mentioned before, there is one particular area of EWFNs that needs to be extended in order to be able to represent BPEL process models. This required extension only concerns the types and structure of the tuples communicated. No other modifications to the generic EWFN model are required. The extensions are: (i) A general distinction between tuples representing control and data flow. (ii) Control flow tuples are further separated into tuples representing (positive) control flow or *dead-path* [LR00]. (iii) A mechanism to represent the `scope` nesting and to enforce proper separation of variables in cases of dynamically created scopes, e.g. in a `forEach` activity.

In WS-BPEL, a join is always *synchronizing*, i.e. the execution engine waits until the state of all incoming links is known (either the control flow has arrived or the link has been marked as "dead", meaning that control flow will never be passed through this link). Note that this behavior is independent of the BPEL *join condition* on the particular join node: even if the join condition is a disjunction of the states of all incoming links, it is still required that the state of each incoming link – positive control flow or dead path – is known before the join can be executed. In our model, this information is carried by the control flow (or `CF`) tuple. A join activity therefore can only determine its status once tuples from each incoming branch are available. When this information is present and the join condition over the states of all incoming links evaluates to `true`, the activity performs its business-logic and emits a control flow tuple in order to pass control flow on to the next activity. A join condition evaluating to `false` causes a dead-path (or `DP`) tuple to be passed on to each immediately

following activity to facilitate Dead-Path-Elimination (DPE) [LR00].

Definition 37 ($\text{EWFN}_{\text{BPEL}}$). *$EWFN_{BPEL}$ is a tuple*

$$EWFN_{BPEL} = \langle \Sigma_{BPEL}, \Phi_{BPEL}, P, T, S, F, X, A, B, I_0, L_{write} \rangle$$

$EWFN_{BPEL}$ is based on $EWFN_{Extended}$, the sets P, T, S, F, X and the functions A, B, I_0 and L_{write} remain unchanged. The differences to $EWFN_{Extended}$ are:

- *Set Σ_{BPEL} contains two categories of tuples: control flow and data tuples. Control flow tuples are further detailed into tuples representing positive (called* `CF` *tuples) or negative (so called* `DP` *tuples) control flow. All tuples contain the* scope context tuple.
- *Set Φ_{BPEL} contains the data types used in BPEL which are described by XML Schema [T+04] by default. The definition of Φ_{BPEL} thus is based on a formalization of XML Schema [KML08].*
- *Set R is modified to contain the new* `sync` *operation. In $EWFN_{BPEL}$, $R = \{read, take, readall, takeall, update, sync\}$*

In the following, we discuss each of these extensions individually.

The *scope context tuple* indicates the hierarchy and identifiers of the `scope` a certain tuple belongs to. It is based on the ideas of the `CF` "history field" which was developed in [Pop08].

Definition 38 (scope context tuple). *A scope context tuple is a list of* scope IDs*, i.e. it has the form $\langle \mathbb{N}, \mathbb{N}, \mathbb{N}, \ldots \rangle$.*

The order of the scope IDs in the scope context tuple contain information about the hierarchy of the scopes a certain tuple resides in. A tuple with context $\langle 5, 3, 9 \rangle$ for instance resides within a scope with ID 9, which itself is nested in scope 3, which again is nested in scope 5, which is the top level scope. A more elaborate example of how this is used to formalize dynamically created scopes will be presented in Section 5.4.2.

Definition 39 (Σ_{BPEL}). *The set Σ_{BPEL} contains the tuples communicated to implement the semantics of a BPEL process.*

$$\Sigma_{BPEL} = \{CF, DATA \times \mathbb{N}, DATA \times \mathbb{N} \times String, \ldots, \epsilon\}$$

Σ_{BPEL} contains two different categories of tuples: (i) control flow tokens $CF = \langle \text{"CF"}, D, \mathbb{N}, \mathbb{N}, \langle \mathbb{N}, \mathbb{N}, \ldots \rangle \rangle$ with $D = \{\text{"pos"}, \text{"neg"}\}$ denoting either positive or negative control flow and (ii) data tokens representing BPEL variables and process metadata. The two integer fields represent process ID and instance ID in order to be able to distinguish between different process models and process instances. The last field represents the context tuple, containing integer fields that denote the nesting of scopes and their IDs.

Data tokens consist of the generic data tuple (denoted as $DATA = \langle \text{"}DATA\text{"}, \mathbb{N}, \mathbb{N}, \langle \mathbb{N}, \mathbb{N}, \ldots \rangle \rangle$) concatenated with variable definitions (in tuple form) from the respective process definition, e.g. $DATA \times \mathbb{N}$ or $DATA \times \mathbb{N} \times$ String in the definition of Σ_{BPEL} above. We represent arbitrary structured data by serializing it (e.g. a tree-based representation such as XML-DOM [LH+04]) into nested tuples. Furthermore, Σ_{BPEL} contains the "empty" tuple ϵ used to denote that actually no tuple is produced.

In the following, a control flow token is called `CF` token if the second field contains "pos" and it is called `DP` token when the second field is set to "neg".

Definition 40 (Φ_{BPEL}). *The set Φ contains BPEL's data types, defined by the formalization of XML Schema presented in [KML08].*

$$\Phi_{BPEL} = XMLstruct$$

BPEL uses XML-Schema (XSD) [T+04] as the default type system, the set Φ_{BPEL} consequently is defined on a formalization of XSD, which was developed as part of the abstract syntax definition of WS-BPEL 2.0 [KML08]. As a result, we define $\Phi_{\text{BPEL}} = \text{XMLstruct}$ with the set XMLstruct representing "the infinite set of XML-Schema types that can be defined using XML-Schema and are included in the process definition" [KML08]. As a result, set Φ_{BPEL} contains XSD built-in types such as `xsd:string` or `xsd:float`, but also all complex

types built using the XSD type definition mechanism that are used in the BPEL process to be implemented.

5.1.1.3 Graphical Notation

In order to graphically represent the EWFN patterns discussed in this section, Figure 5.1 visualizes the operations introduced in Sections 4.5 and 5.1.1 additionally to the generic graphical EWFN notation presented in Section 4.2.6.

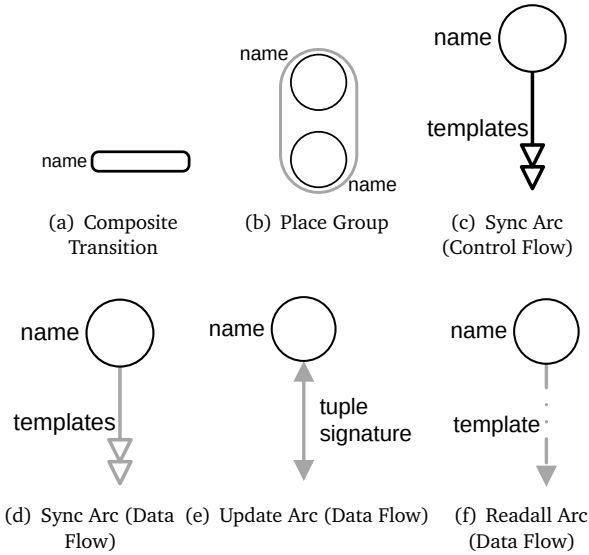


Figure 5.1: Extensions to the Original EWFN Graphical Notation

Throughout this chapter, we distinguish between data and control flow by using different arc colors; black for control flow (i.e. arcs that communicate `CF` tokens) and gray for data flow (i.e. arcs that communicate `DATA` tokens).

Composite transition (Figure 5.1(a)) When formalizing BPEL activities, the resulting EWFNs can get very complex. In order to make the graphical versions of these EWFNs easier to read and understand, we use composite

transitions to represent an EWFN "subnet" in form of a single transition. Whenever we use a composite transition, a separate EWFN pattern will be presented that implements it. The composite transition and the EWFN pattern implementing it will have the exact same interface, i.e. the number of incoming and outgoing arcs and the places they connect to are exactly the same. Basically, the pattern border (dashed line) of the EWFN pattern is collapsed to a single transition which is shown as a composite transition in the higher-level pattern.

Place group (Figure 5.1(b)) In the graphical notation, place groups are used to denote that two or more places are actually one place. In the sequence pattern for instance, the places *Ended* and *Start* of two consecutive activities "overlap" i.e. are just one place and are thus depicted as a place group.

Sync arc (Figures 5.1(c) and 5.1(d)) Arcs with two arrow-heads on one side denote a `sync` arc, which may communicate control or data flow tokens. The `sync` arc follows the semantics of the `sync` operation as sketched in Section 5.1.1.1. In contrast to other operations, `sync` uses multiple templates on a single arc.

Update arc (Figure 5.1(e)) Two-sided arrows represent an `update` operation, which, in the BPEL transformation, is only used for data tuples.

Readall arc (Figure 5.1(f)) An arc with a line that is interrupted by two dots denotes a `readall` operation. Again, in the BPEL transformation, `readall` arcs are only used to communicate data tuples.

If the type of tuple communicated by an arc depends on process state (e.g. one of a list of different fault tuples), the "|" notation as defined in Chapter 4 is used. The arc inscription "$tu_2 \mid tu_3$" for instance means that either tuple tu_2 or tu_3 is communicated by the arc.

5.1.2 Transformation Details

For the transformation, we represent each BPEL activity as a separate pattern. BPEL's static validation rules are not considered for the transformation, a syntactically correct BPEL process is expected as input. BPEL consists of nested blocks; the EWFN patterns presented in the following represent these blocks and can be nested in the same way as their BPEL counterparts. The graph part of BPEL (represented by the `flow` activity and links) is modeled directly, links are again represented by patterns.

Places that are on the border of an EWFN pattern are called *interface places*. These places play a key role in the composition of EWFN patterns. We therefore list all interface places used in the patterns and explain their meaning.

Start* and *Ended The places *Start* and *Ended* may only contain `CF` tuples. A `CF` tuple in place *Start* denotes the arrival of control flow at the EWFN pattern, a `CF` tuple in place *Ended* indicates that control flow has left the pattern and should be passed on to the next activity.

Failed A `DATA` token in place *Failed* indicates that a fault has occurred. This place is defined by the `scope` pattern, i.e. the place *Failed* in a basic activity pattern is always equal to place *Failed* of the pattern representing its enclosing `scope`. The tuple written to place *Failed* represents the fault thrown, e.g. it contains information about the fault type, a fault message, etc.

Variables The place *Variables* is defined by the EWFN `scope` pattern to hold the process variables (in tuple form) defined by the respective `scope`. Whenever it occurs as interface place of an EWFN pattern of a basic activity, it refers to the *Variables* place defined by its surrounding `scope` pattern. The `receive` pattern (Figure 5.2) for instance contains a *Variables* place that is defined by its surrounding scope. To maintain readability, we use a single place for all variables defined in a `scope`. It is clear however that – in the actual deployment of an EWFN – this place should be further decomposed depending on the size and access pattern of the variables stored. In order to provide maximum flexibility and finest distribution

granularity for the partitioning algorithm (see [Wut10]), even a separate *Variable* place for each BPEL variable may be used.

Process Definition This place stores the original process definition, i.e. the involved BPEL and WSDL files. This allows activities to e.g. access join condition expressions or the expressions in an `assign` activity.

Scope State, IMA States These places persist various state created during process execution. The place *Scope State* for instance contains a tuple that is updated when the state of its defining `scope` changed, e.g. from "running" to "ended". The names of the states used in this tuple comply to the ones defined in [Ste08]. Moreover, this place stores information about the order of completion of direct child scopes, which is important to calculate the default compensation order of compensation handlers (cf. Section 5.4.5.1). The place *IMA States* is used by a `scope` to maintain information about the state of inbound message activities (IMA) [A$^+$07] it contains.

Terminate* and *Terminated Places *Terminate* and *Terminated* are used by scopes to force termination of child scopes and for child scopes to signal completion of forced termination to their parent `scope`. A `CF` token in place *Terminate* thus signals a child `scope` to initiate forced termination. After completion, the child `scope` puts a `CF` token into place *Terminated*.

Faults The place *Faults* is used to buffer faults to be potentially forwarded by the `rethrow` activity. It contains `DATA` tuples that represent the information carried by a BPEL fault.

Stop All* and *Stopped All The places *Stop All* and *Stopped All* are used to communicate with a middleware component that implements the "Stop Pattern" [Sta05], i.e. to remove all control flow tokens (`CF` and `DP`) from a running process instance or `scope`. In a `scope`, the places triggering Stop Pattern functionality are called *Stop Scope* and *Stopped Scope*). In related work, the generic functionality behind these places is called "Cancellation Patterns" [vdAADtH04].

TerminateScope* and *ExitProcess The stop pattern, i.e. removing all control flow tokens to halt a process instance is only one part of a two step mechanism to stop a process instance. Before removing the tokens, an activity such as `wait` or `receive` might already be active and thus will not be canceled by removing control flow tokens. For this case, the places *TerminateScope* and *ExitProcess* tell – again through middleware mechanisms such as a multicast – involved activities to immediately stop execution. Details about how these operations are implemented on the middleware level are documented in [Wut10].

The actual mechanism to compose a process from the presented EWFN patterns is based on the principle of *place-overlapping*[1], e.g. the interface places *Ended* and *Start* of two consecutive activities in a sequence overlap, and thus are represented by one single place when composed. Similarly, places from a surrounding `scope` overlap with the interface places of the same name used on the basic activity patterns. Examples of these kinds of places are *Failed*, *Variables*, *PartnerLink* and *Correlation Sets*.

Note that not all interface places are shown on every EWFN pattern. For instance, not every pattern contains a *Failed* place. This is because an EWFN only shows places that it directly interacts with. If the particular activity does not throw a fault, it does not show the *Fault* place (implemented by the surrounding scope) on its list of interface places. Generally, interface places are only displayed on the border of the EWFN pattern if information they contain is accessed or updated. Also, the alternative of not producing a token at all – indicated by the ϵ tuple on arc inscriptions – is not explicitly drawn. All arc inscriptions implicitly are concatenated with "$| \ \epsilon$".

The description of each BPEL activity and its corresponding EWFN pattern is structured into three parts: (i) a brief overview of the BPEL activity itself, (ii) a figure and description of the resulting EWFN and (iii) a textual version of the "token game" with the EWFN to show how different situations such as arrival of a `CF` or `DP` token are implemented.

[1] A similar mechanism was proposed in [KS09] to compose different interaction patterns executed with tuplespaces.

5.2 Basic Activities

Basic activities are the basic building block of a business process. They consist of activities for Web service interaction, namely `receive`, `reply` and `invoke` and variable manipulation – `assign`. Plus functionality for explicitly raising process faults (`throw`), pausing process execution for a specific amount of time (`wait`), doing nothing (`empty`) and to explicitly end the business process (`exit`).

5.2.1 Receive

The BPEL `receive` activity is one of the activities that enables Web service interaction. More specifically, it is an *inbound message activity* (IMA) that defines the WSDL operation it expects to be invoked by the partner through the attributes `partnerLink`, `portType` and `operation`.

There are two different types of `receive` activities from an EWFN point of view, depending on the value of the attribute `createInstance`. In the case of `createInstance="yes"`, there is no *Start* place since there are no control flow dependencies on other parts of the process; the received message alone is the trigger to create a new process instance. Additional checks are required to avoid conditions where messages that share the same *correlation-set* would trigger creation of the same process instance multiple times. Consequently, this section distinguishes between two variants of `receive` and presents an EWFN pattern for each.

Figure 5.2 presents the EWFN form of the BPEL `receive` activity with attribute `createInstance="no"`, i.e. it does not create an new process instance upon reception of a message and therefore has control flow dependencies on other activities expressed through the *Start* place. The corresponding tuple and template definitions of Figure 5.2 are listed in Table 5.1. In the following, we discuss the semantics of the `receive` EWFN based on two scenarios: reception of a `CF`, or `DP` control flow token in place *Start*.

`CF` **token in place *Start*** A `CF` token in place *Start* together with matching tuples for te_1 and te_2 enable transition t_1 that checks for (i) conflict-

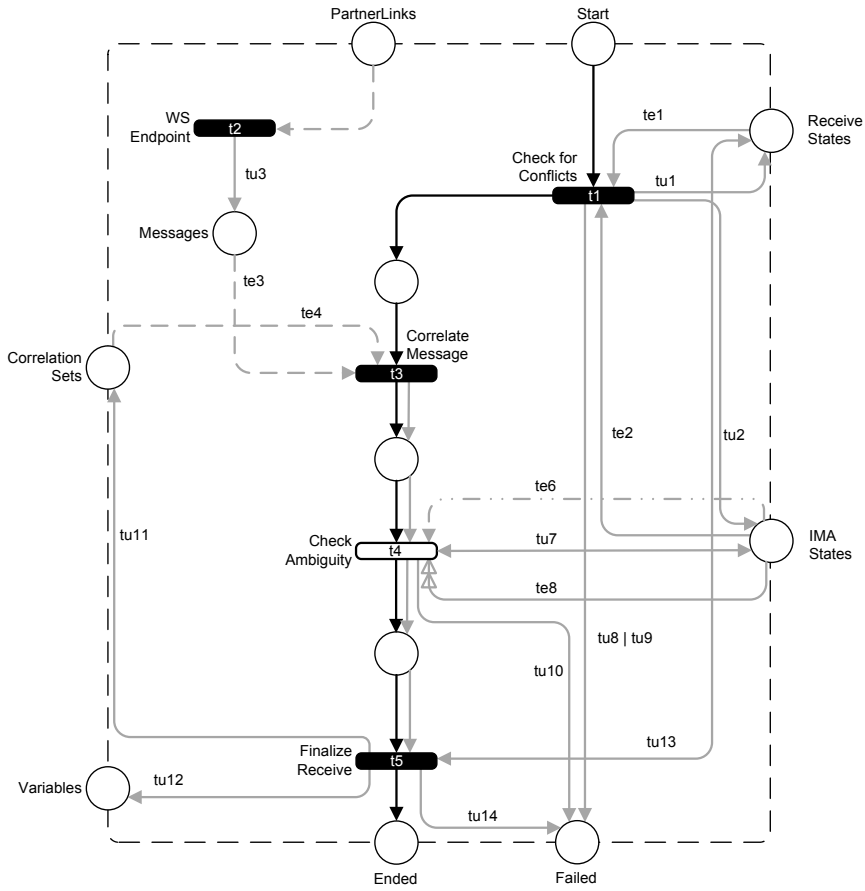


Figure 5.2: EWFN for `receive`

ing receive and (ii) conflicting request operations in the current process instance. If any of those conflicts are detected, t_1 produces a DP token as control flow output (for later synchronization of scope termination) and a corresponding fault tuple (tu_8 or tu_9, signaling either `bpel:conflictingReceive` or `bpel:conflictingRequest`) on place *Failed*. Templates te_1 and te_2 are used to detect those situa-

tions. te_1 checks if there is a `receive` already active with the same `partnerLink`, `operation` and `correlationSet` by checking if a tuple for te_1 is returned. The check for a conflicting request (i.e. multiple simultaneous open IMAs with the same `partnerLink`, `operation` and `messageExchange`) works similarly: if a tuple for template te_2 is returned, a conflicting IMA is currently active and a conflicting request fault must be thrown. Given the current process instance passes both tests, t_1 writes tuples tu_1 and tu_2 to mark (state "open") the Web service interaction as explicitly owned by this particular `receive` activity.

Transition t_2 represents the Web service endpoint and produces tuples representing the received message into place *Messages* upon reception of a Web service call. Note that t_2 may fire any time (more precisely, it fires on each incoming Web service request), since it does not have any incoming arcs except for the information it reads from place *PartnerLinks*, which is deployment information and thus always available. Received messages are buffered in place *Messages* independent of the control flow state of the process instance. The `receive` activity blocks if no matching message has been received yet, and continues immediately if a matching message is already available in place *Messages*.

Transition t_3 fires as soon as control flow arrives, a matching Web service message via te_3 and a *correlation set* matching te_4 are available. It then passes control flow and the received message on to subsequent transitions (depicted using the data flow arcs going from t_3 through t_4 to t_5).

Transition t_4 carries out ambiguity checks for active `receive` operations that match the same message. Since this is a quite complex task, a composite transition in this EWFN only shows the "interface" (i.e. the operations it carries out plus associated templates and tuples) required for that task; its precise semantics are detailed by the EWFN behind this transition, which is presented in Figure 5.3.

Transition t_5 finalizes the `receive` and fires when a `CF` token and the message is available on its input place. It then updates the receive state via tuple tu_{13} (state "closed"), writes the requested parts of the message

to the tuple that represents the BPEL process variable (tu_{12}) and possibly initializes the specified correlation set (if `initiate="yes"` or `"join"`) using tu_{11}. Also, t_5 checks the correlation consistency and correlation initiation constraints and signals a `bpel:correlationViolation` fault via tu_{14} if any of those fail. Finally, it passes control flow on to successor activities via place *Ended*.

`DP` **token in place *Start*** A `DP` token is forwarded to place *Ended* through each transition on the way. A `DP` token in place *Start* therefore always results in an `DP` token in place *Ended*.

Table 5.1: Template and Tuple Definitions for Figure 5.2 and 5.3

Symbol	Operation	Definition
te_1	take	⟨partnerLink, operation, correlationSet, "open"⟩
tu_1	write	⟨partnerLink, operation, correlationSet, "open"⟩
te_2	take	⟨partnerLink, operation, messageExchange, "open"⟩
tu_2	write	⟨partnerLink, operation, messageExchange, "open"⟩
te_3	read	⟨messageID, message⟩
tu_3	write	⟨messageID, message⟩
te_4	read	⟨correlationSet⟩
te_6	readall	⟨partnerLink, operation, *, "open"⟩
tu_7	update	⟨partnerLink, operation, correlationSet, "open", messageID⟩
te_8	sync	⟨⟨partnerLink, operation, * "open", *⟩, ⟨partnerLink, operation, *, "open", *⟩, ...⟩
tu_8	write	⟨fault: conflictingReceive⟩
tu_9	write	⟨fault: conflictingRequest⟩
tu_{10}	write	⟨fault: ambigiousReceive⟩
tu_{11}	write	⟨correlationSet⟩
tu_{12}	write	⟨variable⟩
tu_{13}	update	⟨partnerLink, operation, correlationSet, "closed"⟩
tu_{14}	write	⟨fault: correlationViolation⟩

Figure 5.3 details the semantics of composite transition t_4 in Figure 5.2, corre-

sponding template definitions are listed in Table 5.1. It implements the runtime test for ambiguous `receive` operations. A `receive` is called ambiguous "if a business process instance simultaneously enables two or more IMAs for the same `partnerLink`, `portType`, `operation`, but different `correlationSets`, and the correlations of multiple of these activities match an incoming request message" [A+07].

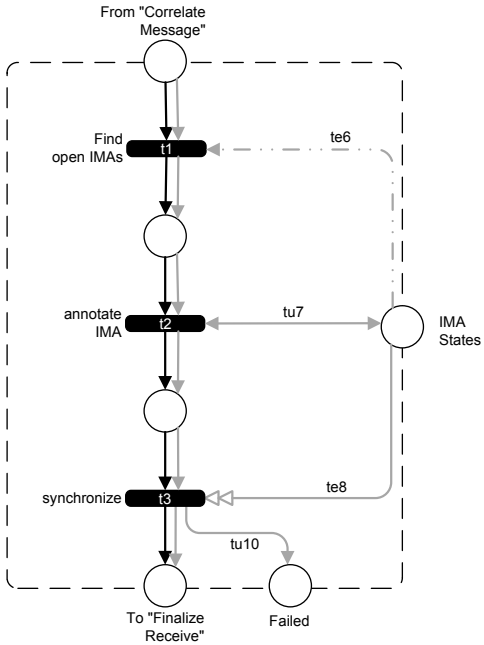


Figure 5.3: "Check Ambiguity" Composite Transition of the `receive` EWFN

A three-step procedure is used to detect an ambiguous `receive`: (i) t_1 finds all open IMAs with the `partnerLink` and `operation` of the current `receive` using a `readall` with template te_6. It passes the information about the number n of returned tuples on to transition t_3 (depicted by the data arc) to construct as many templates for the `sync` operation as there were open IMAs. Next, (ii) transition t_2 updates the receive's IMA state with the messageID of the received

message using tu_7. Finally, transition t_3 issues a `sync` operation over the n IMAs returned by the `readall` operation from t_2 to block execution until all open IMAs detected in step (i) received a message and consequently updated their IMA state using tu_7. Transition t_3 then checks if there is a pair of IMAs in the result of the `sync` operation that contains equal messageIDs. If this is the case, an `ambiguousReceive` fault is thrown using tu_{10} and a `DP` tuple is produced on the control flow arc. Otherwise, control flow continues normally indicated by a `CF` token.

Figure 5.4 shows the EWFN for `receive` with `createInstance="yes"` (i.e. the `receive` is a *start activity*), corresponding tuple and template definitions are listed in Table 5.2. The difference compared to the non-instantiating `receive` is that there are no control flow dependencies from preceding activities, i.e. the pattern is missing a *Start* place and – since the `receive` creates a new process instance upon reception of a message – it contains a transition (t_6) that initializes the surrounding `scope`. Also, additional checks have to be carried out in order to ensure that concurrently arriving messages that match the same correlation information do not create two separate process instances. In the following, we discuss the EWFN pattern based on two scenarios: (i) two concurrent messages arrive with equal correlation information in a process model with two start activities sharing the same `correlationSet` and (ii) a single message arrives and creates a new process instance.

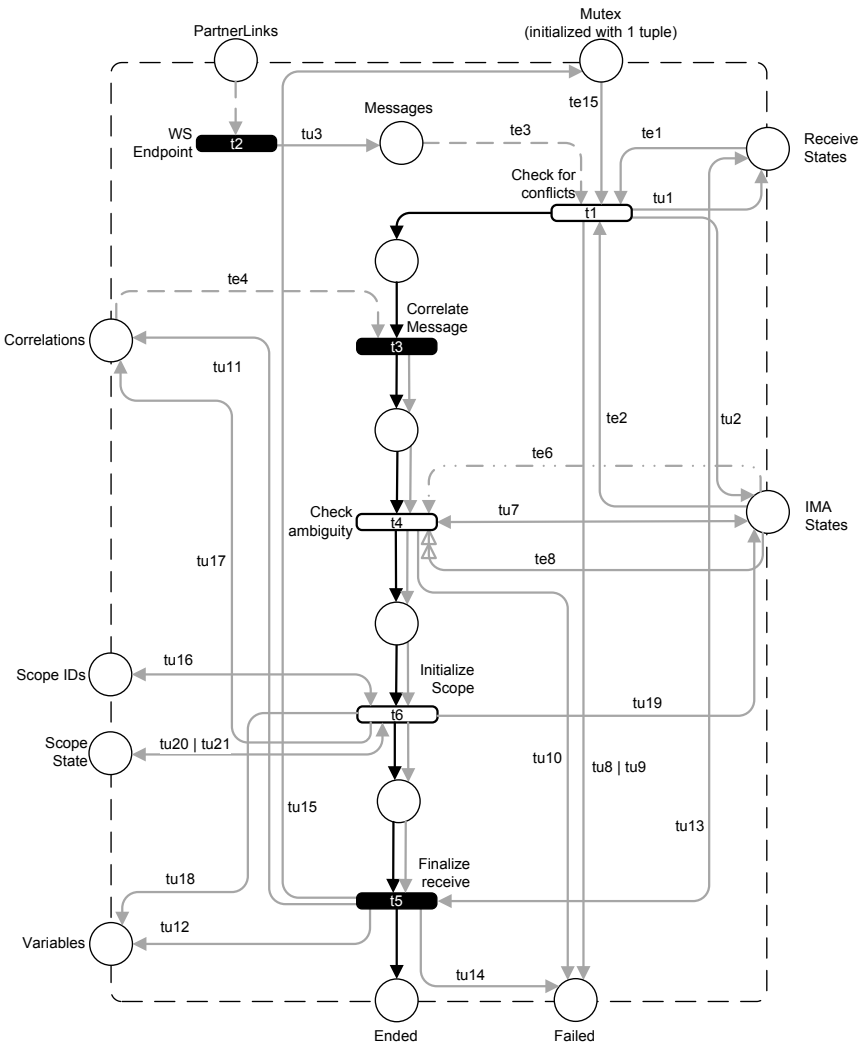


Figure 5.4: EWFN for the `receive` Activity with `createInstance="yes"`

Two concurrent messages sharing the same correlation information arrive

According to [A+07], it is permissible to have multiple start activities,

however, the initial start activity (i.e. the activity that caused creation of a new process instance) must complete execution before any other start activities are allowed to execute. Special care has to be taken in the case where two or more concurrent messages (called *concurrent initial set*) that share the same correlation information arrive at the process engine. In this case, one of the messages triggers the initial start activity that creates a new process instance, all other members of the concurrent initial set are correlated to their corresponding `receive` activities after the initial start activity has completed. They thus do not cause other process instances to be created.

Since the initial start activity forces other start activities to suspend execution until it is finished, transition t_1 is slightly more complex – and therefore modeled as composite transition – compared to t_1 in the non-start activity case (Figure 5.2). Its detailed semantics are depicted in Figure 5.5, corresponding tuple and template definitions can be found in Table 5.2. The pattern uses a mutex tuple to enforce isolated execution of the initial start activity. Transition t_1 of Figure 5.5 is therefore only enabled if a matching message (via te_3) and the mutex tuple (via te_{15}) is available. Then, the receive- and IMA state checks known from the non-start activity `receive` are carried out by transition t_2. Given all checks are passed successfully, control flow continues to t_3 in Figure 5.4 that is responsible to correlate the received message. Since the message itself triggers a new process instance, the `correlationSet` is not initialized at this point and a new process instance ID must be created by transition t_3 and added to the control flow tuple. Control flow then continues to t_4, t_6 and t_5 which do message ambiguity checks, initiation of the `correlationSet` and other finalization tasks as in the non-start activity case. Transition t_6 initializes the surrounding `scope`; it is a composite transition that is equal to transition t_1 in the `scope` EWFN (see Figure 5.23), and is detailed in Figure 5.24. Finalization of the `receive` includes one additional step: the mutex tuple is given back in order to signal that the initial start activity has completed execution.

The second message of the concurrent initial set is held back until this point. Now that the new process instance was created, the `receive` this message was targeted at is able to acquire the mutex tuple and can proceed. Transition t_3 in this case correlates the message to the newly created process instance and proceeds normally.

A single message arrives If the concurrent initial set contains only one message, the EWFN behaves exactly like it is processing the first message of a larger concurrent initial set. It acquires the mutex tuple, creates a new process instance and possibly initializes a correlation set so that other messages can be correlated to the newly created process instance.

It is important to note that there is a mutex tuple per process model, i.e. all start activities from a certain process model share a single mutex, effectively serializing process instance creation. Another interesting fact about the start activity `receive` pattern is that parts of it use "anonymous" `CF` tokens. On reception of a Web service message that needs to be checked and possibly correlated, `CF` tuples from (and within) t_1 to t_3 contain no *process instance ID* and thus are anonymous. It is transition t_3 that either creates a new process instance ID or receives it through message correlation and finally adds the ID to the `CF` tuple when passing control flow on to t_4. Creating a new process instance ID means to add a unique number to the respective field of the `CF` tuple that will be written to indicate that control is passed on to the next transition.

5.2.2 Reply

The `reply` activity belongs to the set of WS-BPEL interaction activities and is responsible for sending a response to a request previously accepted through an IMA, a `receive` for instance, in order to complete a request-response operation.

Figure 5.6 presents the EWFN for the `reply` activity, corresponding template and tuple definitions are listed in Table 5.3. The most notable transition in this pattern is t_4, resembling the Web service endpoint. Its task is the opposite of t_2 from the `receive` EWFN (Figure 5.2), where it was responsible to *receive* a

Table 5.2: Template and Tuple Definitions for Figure 5.4 and 5.5

Symbol	Operation	Definition
te_1	take	⟨partnerLink, operation, correlationSet, “open”⟩
tu_1	write	⟨partnerLink, operation, correlationSet, “open”⟩
te_2	take	⟨partnerLink, operation, messageExchange, “open”⟩
tu_2	write	⟨partnerLink, operation, messageExchange, “open”⟩
te_3	read	⟨messageID, message⟩
tu_3	write	⟨messageID, message⟩
te_4	read	⟨correlationset⟩
te_6	readall	⟨partnerLink, operation, *, “open”⟩
tu_7	update	⟨partnerLink, operation, correlationSet, “open”, messageID⟩
te_8	sync	⟨⟨partnerLink, operation, * “open”, *⟩, ⟨partnerLink, operation, *, “open”, *⟩, . . . ⟩
tu_8	write	⟨fault: conflictingReceive⟩
tu_9	write	⟨fault: conflictingRequest⟩
tu_{10}	write	⟨fault: ambigiousReceive⟩
tu_{11}	write	⟨correlationSet⟩
tu_{12}	write	⟨variable⟩
tu_{13}	update	⟨partnerLink, operation, correlationSet, “closed”⟩
tu_{14}	write	⟨fault: correlationViolation⟩
te_{15}	take	⟨“mutex”⟩
tu_{15}	write	⟨“mutex”⟩
tu_{16}	write	⟨scopeID⟩
tu_{17}	write	⟨scopeID, variableName, initialValue⟩
tu_{18}	write	⟨partnerLink, operation, correlationset, “open”⟩
tu_{19}	write	⟨partnerLink, operation, messageExchange, “open”⟩
tu_{20}	write	⟨scopeID, “running”⟩
tu_{21}	write	⟨scopeID, “dead”⟩

Web service message. Here, it is responsible to *send* the message to the Web service partner identified by the `partnerLink`. In the following, we discuss the `reply` EWFN pattern based on two scenarios depending on whether a `CF`

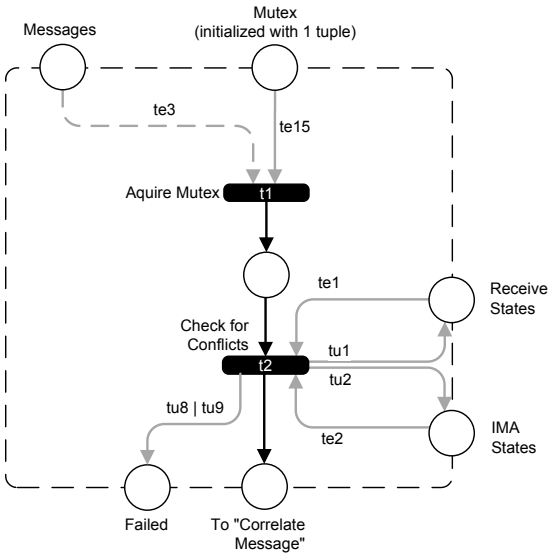


Figure 5.5: “Check for Conflicts” Composite Transition of the `receive` EWFN

or a `DP` token arrives in place *Start*.

Table 5.3: Template and Tuple Definitions for Figure 5.6

Symbol	Operation	Definition
te_2	read	⟨partnerLink, operation, messageExchange, *⟩
tu_3	write	⟨fault:missingRequest⟩
tu_4	write	⟨fault:correlationViolation⟩
te_5	read	⟨correlationSet⟩
tu_5	write	⟨correlationSet⟩
te_6	read	⟨variable⟩
tu_7	update	⟨partnerLink, operation, messageExchange, “closed”⟩

`CF` **token in place *Start*** A `CF` token in place *Start* triggers t_1, which is responsible to check via te_2 if there is an open IMA available that matches the

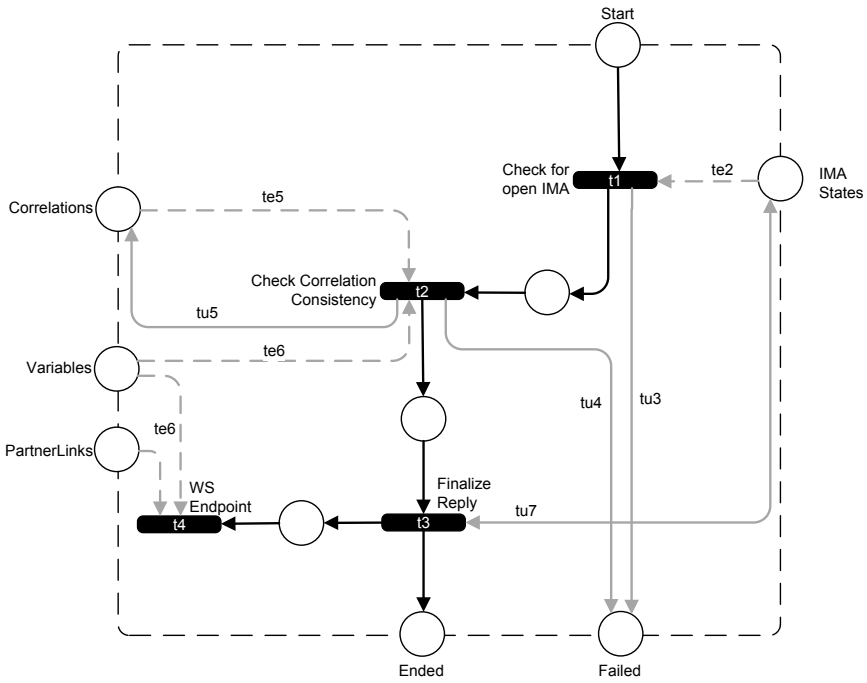


Figure 5.6: EWFN for `reply`

reply's `messageExchange`. If this is not the case, a `missingRequest` fault is written to place *Failed* and a `DP` token is produced on the outgoing control flow arc (note that the IMA state is *read* using te_2, not taken, allowing for the detection of orphaned IMAs later on). Otherwise, control flow is passed to t_2 in order to initiate the `correlationSet` (if requested) and carry out correlation consistency checks according to [A+07]; a `correlationViolation` fault is thrown using tu_4 if the checks are not passed. Transition t_2 therefore reads (te_5) and writes (tu_5) `correlationSet` tuples and accesses values from the reply's data variable (te_6). Control flow then goes on to t_3 responsible to finalize the `reply`. Finalization includes updating the IMA state to "closed" via

tu_7 and triggering the Web service interaction upon completion that finally sends out the message to the partner service. Sending out the message involves reading the reply's data variable and checking place *PartnerLinks* for the actual partner to send the message to. The pattern signals completion of the `reply` by writing a `CF` token into place *Ended*.

`DP` **token in place *Start*** A `DP` token is forwarded to place *Ended* through each transition on the way. A `DP` token on place *Start* therefore always results in an `DP` token in place *Ended*.

5.2.3 Invoke

The `invoke` activity belongs to the set of WS-BPEL interaction activities and is used to call a Web service operation described in WSDL and referenced by a `partnerLink`. Operations can be two-way (e.g. request-response) or one-way (e.g. a notification), corresponding to WSDL 1.2 message exchange patterns [C+07b].

Figure 5.7 shows the EWFN pattern for the `invoke` activity, corresponding template and tuple definitions are listed in Table 5.4. The pattern is basically a combination of the `reply` and `receive` EWFNs without the IMA checks, since a two-way `invoke` involves sending and blocking reception of a message as implemented by these patterns. The IMA checks are left out since there is no open IMA created; the reply is send immediately after the request. In the following, we discuss the `invoke` EWFN pattern based on two scenarios: arrival of a `CF` or a `DP` token in place *Start*.

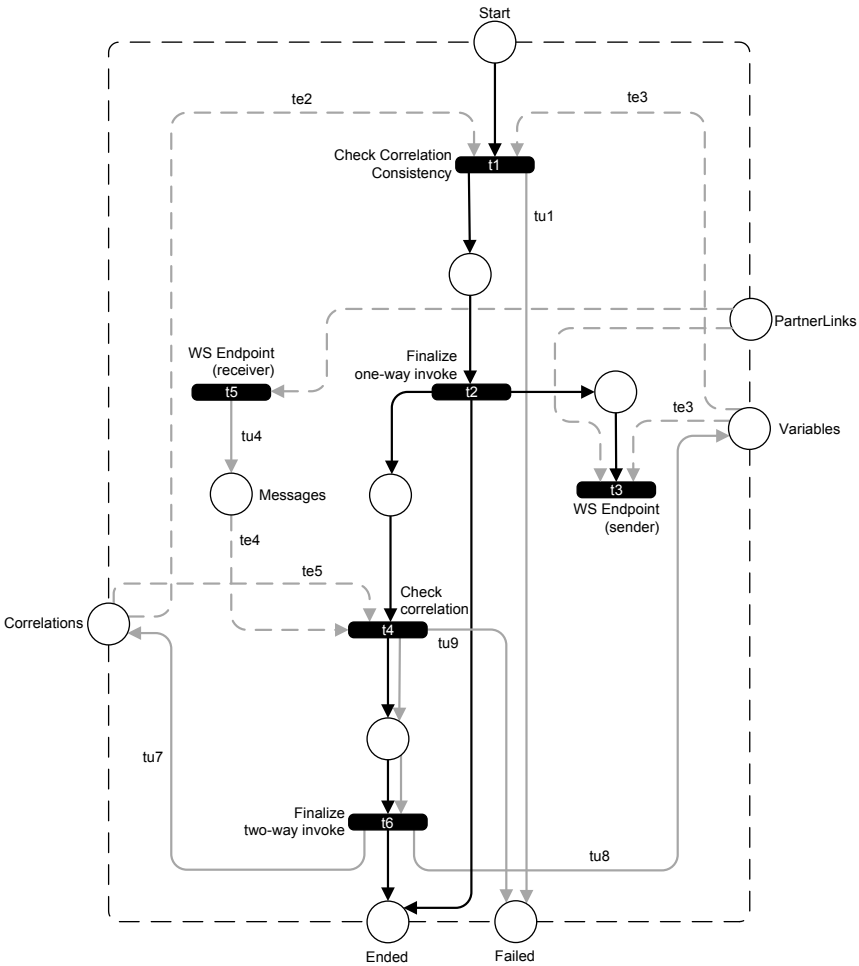


Figure 5.7: EWFN for `invoke`

`CF` **token in place *Start*** A `CF` token in place *Start* together with a matching tuple for te_2 and te_3 trigger execution of t_1 that carries out correlation consistency checks that either result in a `correlationViolation` fault to be thrown (tu_1) or control flow to be passed on to transition t_2. In the

Table 5.4: Template and Tuple Definitions for Figure 5.7

Symbol	Operation	Definition
tu_1	write	⟨fault:correlationViolation⟩
te_2	read	⟨correlationset⟩
te_3	read	⟨variable⟩
te_4	read	⟨message, messageID⟩
tu_4	write	⟨message, messageID⟩
te_5	read	⟨correlationSet⟩
tu_7	write	⟨correlationSet⟩
tu_8	write	⟨variable⟩
tu_9	write	⟨fault:correlationViolation⟩

case of a one-way `invoke`, t_2 finalizes it by passing control flow on to t_3 and place *Ended*. In the case of a two-way `invoke` however, control flow goes on to transition t_4 that correlates received messages and therefore blocks until a matching response is received (via te_4) by the Web service endpoint implemented by transition t_5. After message reception, control flow is passed to t_6 that finalizes the two-way `receive` and initializes the correlation set if required. A `CF` token in place *Ended* finalizes the `receive`.

`DP` **token in place *Start*** A `DP` token is forwarded to place *Ended* through each transition on the way. A `DP` token on place *Start* therefore always results in an `DP` token in place *Ended*.

5.2.4 Assign

The `assign` activity is used to manipulate process variables, e.g. to copy data from one variable to another or to construct and insert new data computed by expressions. It can also be used to copy Endpoint References (EPR) [B+04] to and from `partnerLinks`.

Figure 5.8 shows the EWFN pattern for the `assign` activity, corresponding template and tuple definitions are listed in Table 5.5. The `assign` pattern

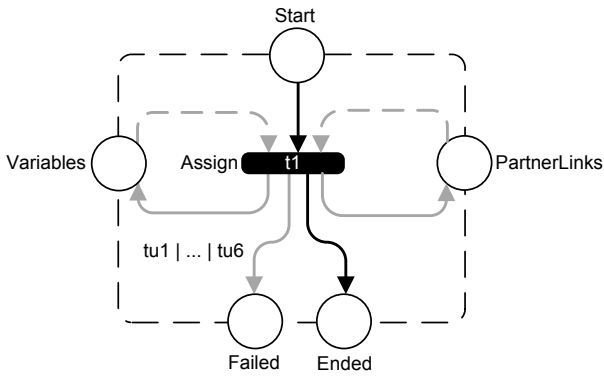


Figure 5.8: EWFN for `assign`

consists of one single transition that implements the required functionality. In the following, we discuss the `assign` EWFN based on two scenarios: arrival of a `CF` or a `DP` token in place *Start*.

Table 5.5: Template and Tuple Definitions for Figure 5.8

Symbol	Operation	Definition
tu_1	write	⟨fault:selectionFailure⟩
tu_2	write	⟨fault:uninitializedPartnerRole⟩
tu_3	write	⟨fault:xsltInvalidSource⟩
tu_4	write	⟨fault:xsltStylesheetNotFound⟩
tu_5	write	⟨fault:subLanguageExecutionFault⟩
tu_6	write	⟨fault:uninitializedVariableFault⟩

`CF` **token in place *Start*** Depending on the kind of `assign` – whether it reads from a `variable` or from a `partnerLink`, t_1 reads the respective tuple from either place *Variables* or *PartnerLinks*. It then processes the expression if defined and writes the result back to either place *Variables* or *PartnerLinks*. Finally, a `CF` token is written to place *Ended* in order to pass control flow on to the next activity. Various faults may

be thrown during execution of the activity: a `selectionFailure` fault (tu_1) indicates a problem with the to-spec of the `assign` e.g. when a non-lvalue is returned. A `uninitializedPartnerRole` fault (tu_2) indicates a situation where an uninitialized `partnerRole` EPR is accessed. Problems with XSLT [C+99] transformations are signaled by the faults `xsltInvalidSource` (tu_3), `xsltStylesheetNotFound` (tu_4) and `subLanguageExecutionFault` (tu_5). An uninitialized variable is reported by a `uninitializedVariableFault`, communicated as tu_6.

`DP` **token in place *Start*** A `DP` token is forwarded to place *Ended* through each transition on the way. A `DP` token on place *Start* therefore always results in an `DP` token in place *Ended*.

5.2.5 Throw

The BPEL `throw` activity is used to signal internal process faults explicitly, e.g. to trigger the fault handler of the enclosing `scope` that populates the fault information to other services. The fault must be identified by a QName, further information such as error text or number may be provided using information from a process variable.

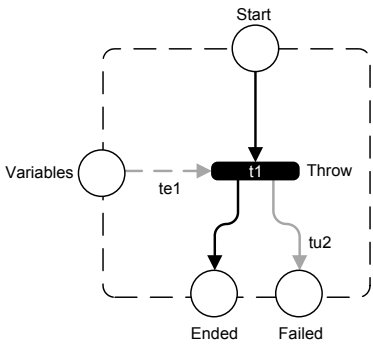


Figure 5.9: EWFN for `throw`

Figure 5.9 shows the EWFN pattern for the `throw` activity, corresponding template and tuple definitions are listed in Table 5.6. The `throw` pattern consists of one single transition that implements the required functionality. In the following, we discuss the `throw` EWFN based on two scenarios: arrival of a `CF` or a `DP` token in place *Start*.

Table 5.6: Template and Tuple Definitions for Figure 5.9

Symbol	Operation	Definition
te_1	read	⟨variable⟩
tu_2	write	⟨faultName, faultMessage⟩

`CF` **token in place *Start*** A `CF` token in place *Start* together with a matching tuple for te_1 triggers t_1 that extracts the information from the variable and populates tuple tu_2 that is then written to place *Failed* upon firing. At the same time, a `DP` token is written to place *Ended* to indicate occurrence of a fault to subsequent activities. This token is also used by the fault and termination handlers to synchronize termination of the enclosing `scope` (See Section 5.4.3 for details). If no fault variable is given in the process definition, the only precondition for t_1 is the availability of a `CF` token in place *Start*. Consequently, field `faultMessage` of tuple tu_2 does not contain a value.

`DP` **token in place *Start*** A `DP` token is forwarded directly to place *Ended*.

5.2.6 Wait

The `wait` activity is used to specify a delay of execution until a specific point in time or a certain deadline is reached. The expressions specifying the time to wait or the deadline may depend on process state, e.g. to wait for an interval that is defined during runtime by the result of a previous `invoke` operation.

Figure 5.10 shows the EWFN pattern for the `wait` activity, corresponding template and tuple definitions are listed in Table 5.7. The `wait` pattern consists of one single transition that implements the required functionality. In the

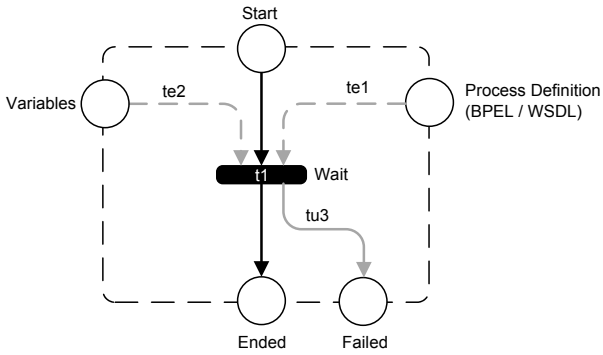


Figure 5.10: EWFN for `wait`

following, we discuss the EWFN based on two scenarios: arrival of a `CF` or a `DP` token in place *Start*.

Table 5.7: Template and Tuple Definitions for Figure 5.10

Symbol	Operation	Definition
te_1	read	⟨wait expression, wait type⟩
te_2	read	⟨variable⟩
tu_3	write	⟨fault:subLanguageExecutionFault⟩

`CF` **token in place *Start*** A `CF` token in place *Start* and matching tuples for te_1 and te_2 enable t_1 that waits the specified amount of time and then writes a `CF` tuple to place *Ended*. In order to execute the expressions that calculate the time to wait, transition t_1 reads variable data from place *Variables* using te_2. The expression itself and the type of wait (`for` or `until`) is read from the process definition using te_1.

`DP` **token in place *Start*** A `DP` token is forwarded directly to place *Ended*.

5.2.7 Empty

The `empty` activity is used to indicate a no-op, i.e. an activity that actually does nothing. This is often useful in `fault handlers` (see Section 5.4.3) to ignore a fault or in a `flow` (see Section 5.4) to introduce an artificial node for adding a join-condition.

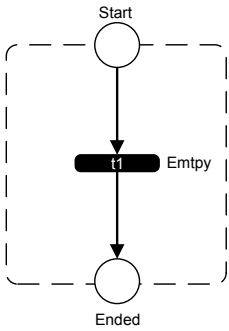


Figure 5.11: EWFN for `empty`

Figure 5.11 shows the EWFN pattern for the `empty` activity. Its pattern consists of one single transition that simply forwards all incoming tokens on place *Start* to place *Ended*.

5.2.8 Exit

The `exit` activity is used to end a process instance; all currently running activities are ended immediately without triggering fault, termination or compensation handling.

Figure 5.12 depicts the EWFN pattern for the `exit` activity. Its pattern consists of one single transition that triggers the two steps required to end a process instance: step 1 removes all control flow tokens (`CF` or `DP`) from *all* places of the process by writing a `CF` tuple to place *Stop All*. Step 2 tells all running activities (e.g. an active `wait`) to immediately cancel execution. In our implementation, both steps are not implemented by tuplespace clients (i.e. transitions), but by the execution middleware itself. Details about this solution

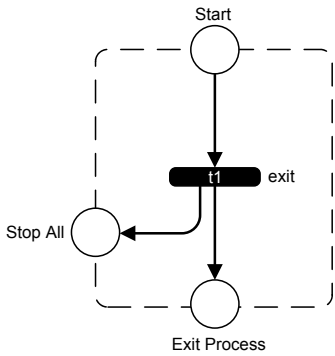


Figure 5.12: EWFN for `exit`

are discussed in the following section.

5.2.9 Process Termination

The related work analysis has shown (cf. Section 2.2.3.1) that the implementation of `scope` semantics, specifically termination, are very hard to implement in a distributed setting. This is due to the "global semantics" of the `exit` activity, i.e. termination requires control flow to stop and *all* activities to cancel execution immediately. A single activity thus has global effect on all other activities in a process.

In related BPEL to Petri net transformations [Loh08, HSS05], termination is implemented on the level of the Petri net by means of a so called "Stop Pattern", each pattern offers a set of transitions that remove all tokens from all places. These transitions are triggered by the availability of a token in a place called "Stop", remove all tokens from the places of the pattern and forward the stop signal along the nesting of the process model up until all "leave nodes" of the process model are reached and ultimately execution has halted. It is clear that a naïve implementation of this pattern may lead to massive problems in the distribution model; effectively, the stop transitions are connected to all places of the process and the forwarding along the paths may introduce unwanted

latencies where parts of the process are still executed while termination is already in progress.

We solve this problem by declaring the EWFN that implements the Stop Pattern "virtual" in the sense that it will be executed by the tuplespace middleware that implements the functionality of places rather than by client applications that implement transitions. As a result, tuple removal logic is pushed down directly to the level where the tuples reside, avoiding unnecessary remote communication.

The problem of canceling all running activities is also solved on the level of tuplespace middleware: a combination of a multicast to all running activities and "blacklisting" of `CF` tokens for the terminated process instance ensures proper termination [Wut10].

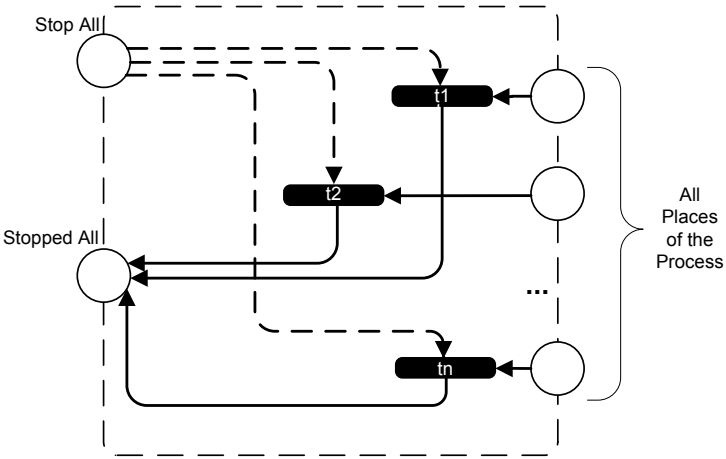


Figure 5.13: "Virtual" EWFN to Stop a Process Instance

Figure 5.13 depicts the virtual EWFN pattern for stopping a process instance. Transitions t_1 through t_n are responsible to remove CF tokens from all places when a `CF` token in place *Stop All* is present. When they fired, they put a token into place *Stopped All* which then contains a token for each removed token in the process. Again, the reader is directed to [Wut10] which presents details on

how the functionality of this EWFN is implemented.

In Section 5.4.3.1, a similar pattern for terminating a `scope` will be presented. The only difference to this pattern is that instead of removing `CF` tokens, they are changed to `DP` tokens in order to ensure that the state of outgoing links (i.e. links that leave the structured activity) are updated correctly.

5.3 Structured Activities

Structured activities are used to compose basic activities into structures that describe the order of how they are executed. Structured activities itself may again hold structured activities, allowing arbitrary complex control flow structures to be created. Three different control flow patterns are supported: (i) sequential activity execution is controlled by the activities `sequence`, `while`, `if`, `repeatUntil` and the serial version of `forEach`, (ii) concurrency and synchronization is provided by the activities `flow` and the parallel variant of `forEach`. Finally (iii), event-controlled deferred choice is implemented by the `pick` activity.

5.3.1 Sequence

The `sequence` activity imposes a sequential order among the activities it contains. A `sequence` completes when the last contained activity completed.

Figure 5.14 depicts the EWFN pattern for the `sequence` activity. Its pattern does not contain any transition, its semantics is implemented by overlapping places of the contained activities. Overlapping, i.e. the composition of two or more places into one single place, is indicated by a gray rounded box around the involved places. Place *Start* of the sequence pattern for instance overlaps (i.e. is the same place) with place *Start* of the first contained activity. As a result, all control flow tokens travel through the pattern in sequential order, passing each contained activity and finally show up on the pattern's place *Ended* when the last contained activity completed. The semantics of `sequence` therefore is implemented purely by composition, no additional transition is involved.

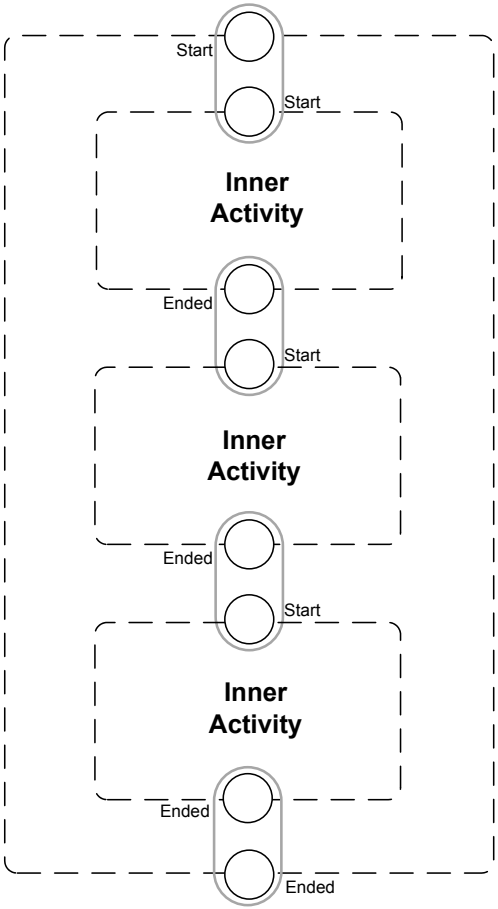


Figure 5.14: EWFN for `sequence`

5.3.2 While

The `while` activity facilitates repetitive execution of activities. Contained activities are executed repeatedly *as long as* the expression in the `condition` element (containing the termination condition of the loop) evaluates to `true`.

Termination semantics are known from many other programming languages, i.e. the condition is checked *before* starting each iteration.

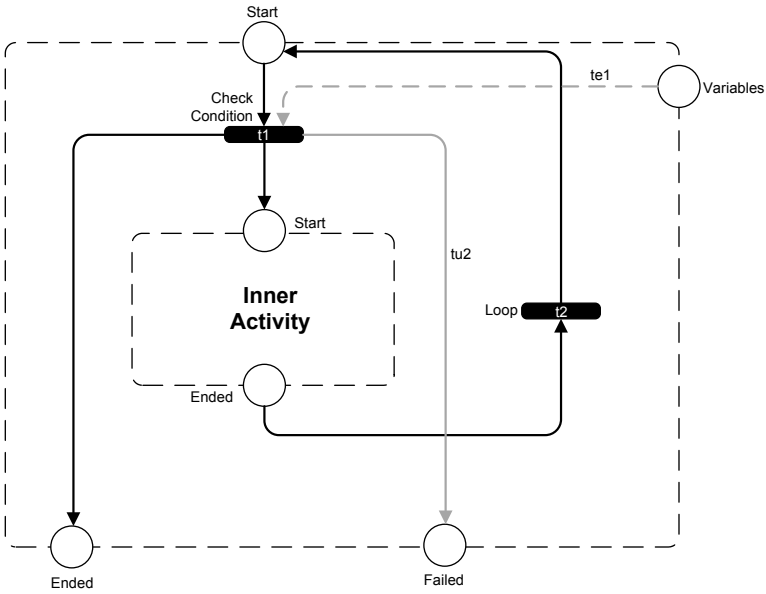


Figure 5.15: EWFN for `while`

Figure 5.15 shows the EWFN pattern for the `while` activity, corresponding template and tuple definitions are listed in Table 5.8. It consists of two transitions, t_1 being responsible for termination condition evaluation and t_2 being a "forwarding transition" to start a new iteration of the loop. In the following, we discuss the `while` EWFN pattern based on two scenarios: arrival of a `CF` or a `DP` token in place *Start*.

Table 5.8: Template and Tuple Definitions for Figure 5.15

Symbol	**Operation**	**Definition**
te_1	read	⟨variable⟩
tu_2	write	⟨fault:subLanguageExecutionFault⟩

`CF` **token in place *Start*** A `CF` token in place *Start* and a matching tuple for te_1 enable transition t_1, responsible for evaluating the termination expression of the loop. Possible faults during expression evaluation are signaled by writing tu_2, indicating a `subLanguageExecutionFault`, into place *Failed* and a corresponding `DP` tuple into place *Ended*. When the result of the condition evaluation is available, t_1 forwards control flow either to the contained activity if the result was `true` or to place *Ended* if the expression evaluated to `false`. When the contained activity completed, transition t_2 forwards the control flow tuple to place *Start* of the pattern in order to start the next iteration.

`DP` **token in place *Start*** A `DP` token is forwarded directly to place *Ended*; link states within the contained activity do not have to be updated since links declared within a `scope` that itself is inside a `while` are not allowed to cross the enclosing `while` activity.

5.3.3 RepeatUntil

Similar to the `while` activity, `repeatUntil` also facilitates repetitive execution of activities. However, contained activities are executed *until* the expression in the `condition` element evaluates to `true`. Also, the condition is checked *after* execution of the contained activity – the body of the loop therefore is executed at least once.

Figure 5.16 shows the EWFN pattern for the `repeatUntil` activity, corresponding template and tuple definitions are listed in Table 5.9. It consists of one single transition that checks the completion condition and either starts a new iteration of the loop or passes control flow on to the next activity. In the following, we discuss the `repeatUntil` EWFN pattern based on two scenarios: arrival of a `CF` or a `DP` token in place *Start*.

`CF` **token in place *Start*** Since the *Start* places of the `repeatUntil` pattern and the inner activity are equal (indicated by the place-group), a `CF` token immediately enables the inner activity. After the first iteration, transition t_1 checks the exit condition of the loop and starts either a new iteration by

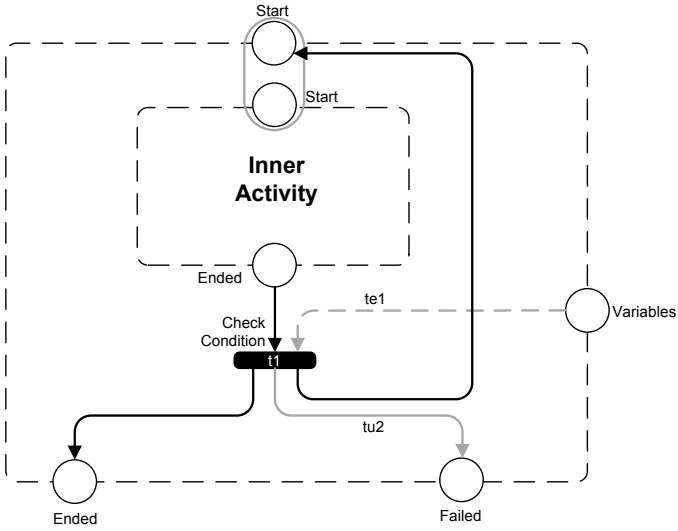


Figure 5.16: EWFN for `repeatUntil`

writing a `CF` tuple to place *Start*, or finalizes the `repeatUntil` activity by writing a `CF` tuple on place *Ended*. If a fault occurred at the inner activity, t_1 receives a `DP` tuple which it forwards to place *Ended* and thus also ends the activity. Possible faults during evaluation of the exit condition are signaled by writing tu_2, indicating a `subLanguageExecutionFault`, into place *Failed* and a corresponding `DP` tuple into place *Ended*.

`DP` **token in place *Start*** Similar to `while`, a `DP` token is forwarded directly to place *Ended* because link states within the contained activity do not

Table 5.9: Template and Tuple Definitions for Figure 5.16

Symbol	Operation	Definition
te_1	read	⟨variable⟩
tu_2	write	⟨fault:subLanguageExecutionFault⟩

have to be updated. Links declared within a `scope` that itself is inside a `repeatUntil` construct are not allowed to cross the enclosing `repeatUntil` activity.

5.3.4 If

The `if` activity allows to define conditional behavior in business processes. It consists of an ordered list of conditional branches that are defined by `if` and optional `elseIf` elements, that are evaluated in the order they appear in the process definition. Branches are executed if their condition evaluates to `true`, otherwise the corresponding else branch, if present, is executed.

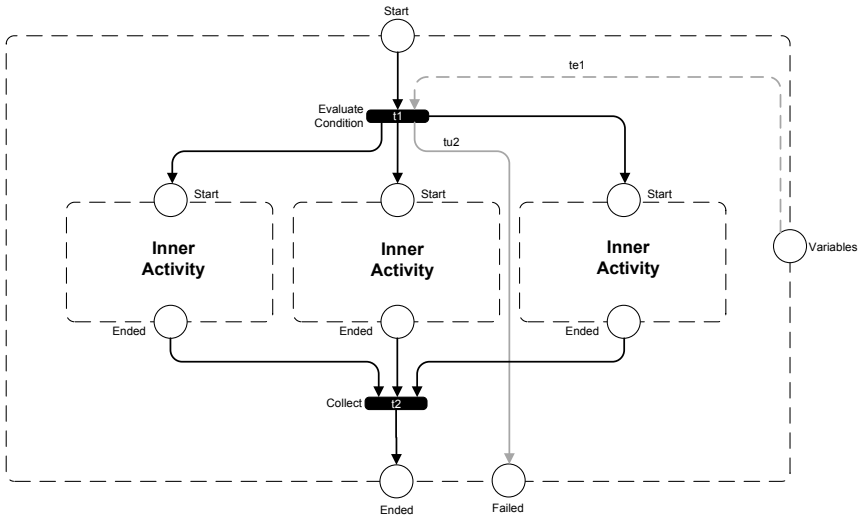


Figure 5.17: EWFN for `if`

Figure 5.17 shows the EWFN pattern for the `if` activity, corresponding template and tuple definitions are listed in Table 5.10. The pattern consists of two transitions, transition t_1 is responsible for evaluating the conditions and forwarding control flow according to the results, t_2 is responsible for synchronizing control flow of contained activities to ensure proper link state. In

the following, we discuss the `if` EWFN pattern based on two scenarios: arrival of a `CF` or a `DP` token in place *Start*.

Table 5.10: Template and Tuple Definitions for Figure 5.17

Symbol	Operation	Definition
te_1	read	⟨variable⟩
tu_2	write	⟨fault:subLanguageExecutionFault⟩

`CF` **token in place *Start*** A `CF` token in place *Start* and a matching tuple for te_1 enable transition t_1 that evaluates the `if` conditions in the order they appear in the process definition. If condition evaluation produces an error, a corresponding fault tuple (tu_2) is written to place *Failed* along with a `DP` tuple that is written to all *Start* places of the inner activities. If no error occurs, a `CF` token is written to the *Start* place of the inner activity that first evaluates to `true`. The remaining contained activities receive `DP` tokens to ensure that link states of flows they may contain are updated. Transition t_2 synchronizes all tokens written to inner activities so that upon completion of the `if` activity, it is guaranteed that the states of links leaving (i.e. crossing the border of) the `if` activity are updated. The output of transition t_2 depends on the execution state of the inner activities: (i) if one of the conditions evaluated to `true`, and no fault was thrown by the inner activity that was executed, t_2 expects a `CF` token from this branch and `DP` tokens from all others. In this case, t_2 writes a `CF` token to place *Ended* of the `if` EWFN. If however a fault was thrown by the inner activity during execution, a `DP` token instead of a `CF` token arrives, causing t_2 to also write a `DP` token to place *Ended*. If none of the conditions of all branches evaluated to `true`, t_2 expects `DP` tokens from all branches and consequently writes a `DP` token to place *Ended* after all of them have arrived.

`DP` **token in place *Start*** `DP` tokens are propagated to all contained activities in order to ensure that their link states (see Section 5.3.6) are updated accordingly. In this case, t_1 writes `DP` tokens to all branches (similar to

the case when none of the conditions evaluated to `true`), t_2 synchronizes their arrival and writes a single `DP` token to place *Ended*.

5.3.5 Pick

The `pick` activity implements *event-controlled deferred choice*, i.e. it waits for the occurrence of exactly one event from a set of events and executes the activity associated with the event received. After processing an event, other events are no longer accepted.

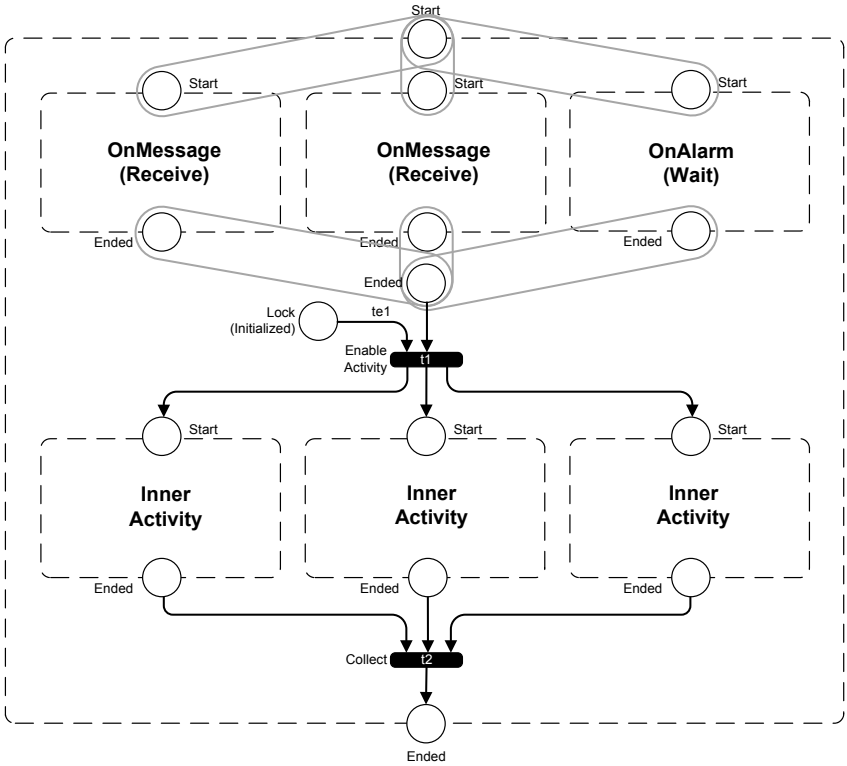


Figure 5.18: EWFN for `pick`

Figure 5.18 shows the EWFN pattern for the `pick` activity, corresponding template and tuple definitions are listed in Table 5.11. The pattern consists of two transitions that (i) ensure that only one event enables the pattern, (ii) dispatch the event to the corresponding activity and (iii) synchronize tokens after the inner activity was executed. In the following, we discuss the `pick` EWFN pattern based on two scenarios: arrival of a `CF` or a `DP` token in place *Start*.

Table 5.11: Template and Tuple Definitions for Figure 5.18

Symbol	Operation	Definition
te_1	take	⟨"lock"⟩

`CF` **token in place *Start*** The event activities (two `receives` and one `wait` in Figure 5.18) are enabled as soon as control flow arrives. In the case of `createInstance="yes"`, the aggregated *Start* places are removed and corresponding `receive` activities are used. As required by the BPEL specification, enabling a branch in `pick` must be followed by disabling all other branches – place *Lock* that is initialized with one tuple ensures that transition t_1 fires only once. t_1 also does the required resolution of race conditions that occur when events are triggered simultaneously, e.g. simultaneous arrival of two messages or a message and a wait-timeout occurring at the same time. These are resolved by removing only one tuple from the aggregated place *Ended* of all event activities. Other `CF` tuples in this place are ignored. Upon firing, transition t_1 enables the branch that was originally triggered by the received event. All other branches receive a `DP` token on their *Start* places. A `DP` token in the aggregate place *Ended* (in the case of a fault thrown by the event activity) cause t_1 to write a `DP` token to all *Start* places. Finally, transition t_2 synchronizes control flow tokens on all *Ended* places of the branches, ensuring proper link-state and to calculate the type of control flow token to be finally written to place *Ended* of the pattern. The logic it employs is equal to transition t_2 in the `if` EWFN.

`DP` **token in place *Start*** `DP` tokens are forwarded by the event activities and then propagated by t_1 to all contained activities in order to ensure that their link states (see Section 5.3.6) are updated. Transition t_1 therefore writes `DP` tokens to all branches, t_2 synchronizes their arrival accordingly and writes a single `DP` token to place *Ended*.

5.3.6 Flow

The `flow` activity provides parallel execution of activities; all activities it contains are enabled concurrently when control flow arrives, the activity completes if all contained activities are completed. By means of links, synchronization relationships can be expressed, resulting in a control flow graph with activities as nodes and links as edges. Without using links, the `flow` activity is used in block-structured modeling to facilitate concurrent execution of activities. The ability to use links with transition and join conditions adds the ability to model graphs instead of blocks allowing us to directly reuse the structure of the graph for the resulting EWFN. Essentially, EWFN patterns accommodate for the block-structured parts of BPEL whereas the graph-structure of `flow` with links can be directly transformed by plugging activities together in the same way they are interconnected through links.

Consequently, the `flow` activity is described in two parts: the first part describes the transformation of the block structured usage of `flow`, Sections 5.3.7 and 5.3.8 describe how the graph structured parts are mapped to EWFNs.

Figure 5.19 shows the EWFN pattern for the block-structured use of `flow`. Here, contained activities are concurrently enabled on arrival of control flow at place *Start*.

`CF` **token in place *Start*** A `CF` token in place *Start* enables transition t_1 that again enables all inner activities that do not have an incoming link. Transition t_2 waits for control flow tokens from all *Ended* places of the concurrently enabled activities, ensuring proper link-state updates. It then calculates the type of control flow token to be finally written to place *Ended*. The logic it employs is equal to transition t_2 of the `if` EWFN.

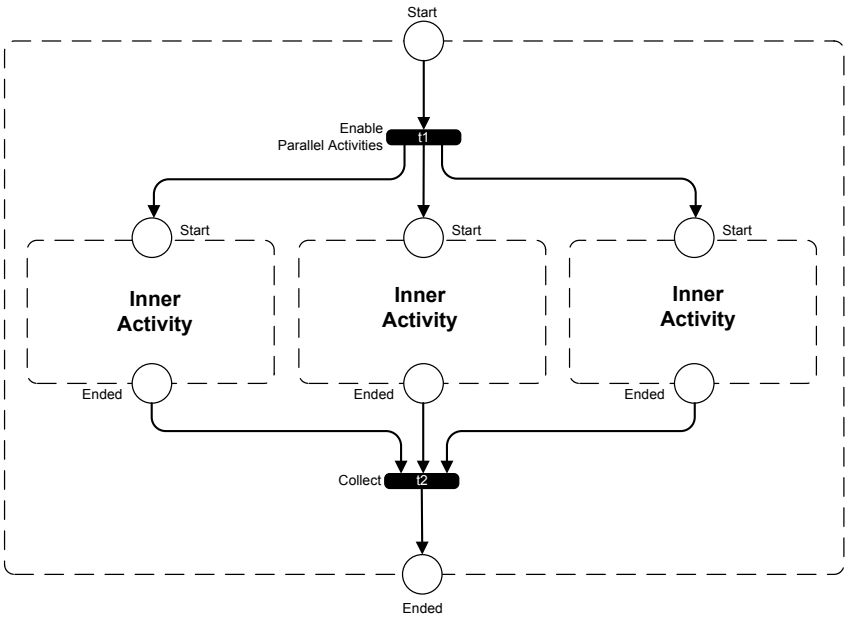


Figure 5.19: EWFN for `flow`

`DP` **token in place *Start*** `DP` tokens are propagated by t_1 to all contained activities in order to ensure that their link states are updated. Transition t_2 synchronizes their arrival accordingly and writes a `DP` token to place *Ended*.

5.3.7 Link-Source

As discussed before, the usage of links within a `flow` leads to a different structure of the resulting EWFN. Instead of creating nested blocks to express process control flow, a graph consisting of activities and links between them is created. Conditional control flow is implemented using transition and join conditions. The former are evaluated at link source, the latter at the link target. We can consequently divide the pattern-implementation of a link into two parts:

the link-source part is discussed in this section, the link-target part is described in Section 5.3.8. Two activities that are connected through a link each receive an EWFN "stub" at pattern end in the case of link-source and at pattern start in the case of link-target, i.e. the *Start* place of pattern link-source equals to the place *Ended* of the source activity and place *Ended* of the link-target pattern is equals to place *Start* of the target activity.

Note that links can *leave*, *enter* or *cross* boundaries of structured activities, functionality that is also covered by their EWFN pattern equivalents. This is implemented by following a construction rule which is based on the observation that the concept of passing tokens is independent from the nesting structure of the patterns. The functionality of scope-crossing control flow passing is thus implemented by passing the token to the corresponding target-link pattern counterpart, independent from the constructs (or the nesting level) that are enclosing it.

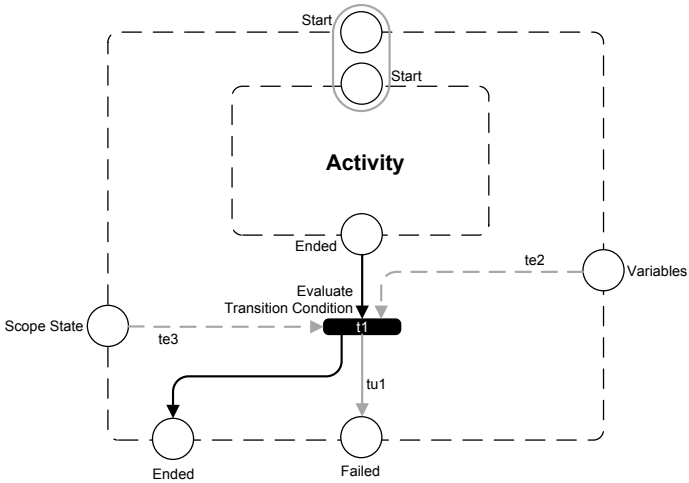


Figure 5.20: EWFN for Link Sources

Figure 5.20 shows the EWFN pattern that is attached to the end of an activity if it is the source activity of a link, its template and tuple definitions are listed in Table 5.12. Its main purpose is to evaluate the transition condition and

generate corresponding control flow tuples, depending on the result of the condition evaluation. Multiple outgoing links (to implement a fork or split) are handled by transition t_1 by evaluating the conditions and writing a control flow tuple to place *Start* of each of the matching link-target EWFNs. Note that in this case, the EWFN pattern has a number of *Ended* places – a separate place for each link. In the following, we discuss the EWFN pattern based on two scenarios: arrival of a `CF` or a `DP` token in place *Start*.

The read arc with te_3 is used in the very special case when the activity of the link source is placed within an *isolated* `scope` (cf. Section 5.4.7). According to [A+07], "the status of links leaving an isolated `scope` will not be visible at the target until the `scope` completes". In this case, transition t_1 only fires when the state of its enclosing `scope` has changed to "ended".

Table 5.12: Template and Tuple Definitions for Figure 5.20

Symbol	Operation	Definition
tu_1	write	⟨fault:subLanguageExecutionFault⟩
te_2	read	⟨variable⟩
te_3	read	⟨scopeID, "ended"⟩

`CF` **token in place *Start*** A `CF` token in place *Start* enables the contained activity that writes a `CF` tuple on its place *Ended* upon completion. This triggers transition t_1 that evaluates the transition condition of the link and writes a tuple to place *Ended* corresponding to the result of the evaluation: `CF` or `DP`. A fault during condition evaluation is signaled by writing a `DP` tuple to place *Ended* and tuple tu_1 to place *Failed*. If the contained activity writes a `DP` tuple to place *Ended*, transition t_1 forwards this tuple to place *Ended* of the pattern.

`DP` **token in place *Start*** `DP` tokens are forwarded by the contained activity (that also might update link states if it contains a `flow`) to place *Ended*, read by t_1 and forwarded to place *Ended* of the pattern.

5.3.8 Link-Target

The link-target pattern is the counterpart of the link-source pattern, its *Start* place therefore is equal to the *Ended* place of one or more matching link-sources. Its main task is to synchronize control flow and to evaluate the join condition over the states of all incoming links.

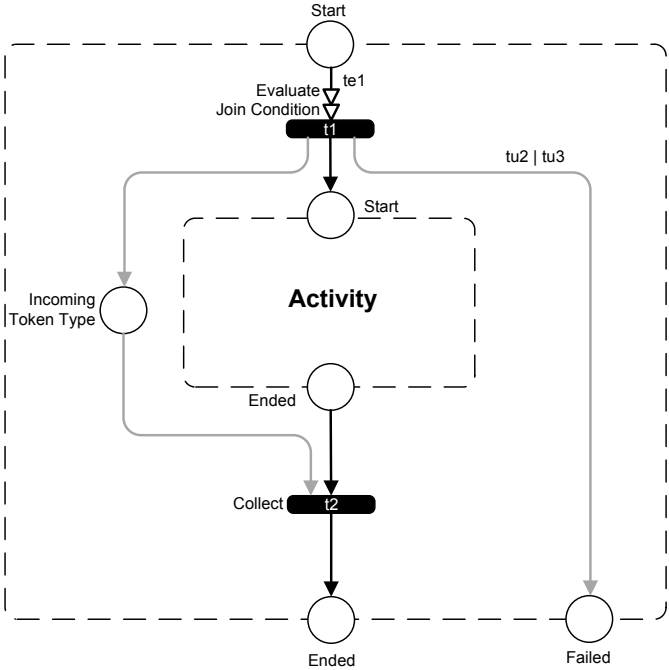


Figure 5.21: EWFN for Link Targets

Figure 5.21 shows the EWFN pattern that implements the functionality for join evaluation around the target activity for one or more links. Template and tuple definitions used are listed in Table 5.13. In the following, we discuss the EWFN pattern based on two scenarios: arrival of a `CF` or a `DP` token in place *Start*.

`CF` token in place *Start* As the *Start* place is an aggregation of all *Ended* places

Table 5.13: Template and Tuple Definitions for Figure 5.21

Symbol	Operation	Definition
te_1	sync	as many control flow tuples as there are incoming branches
tu_2	write	⟨fault:subLanguageExecutionFault⟩
tu_3	write	⟨fault:joinFailure⟩

of matching link-source EWFN patterns, it receives as many `CF` tuples as there are incoming links. Consequently, transition t_1 employs a *sync* operation that blocks until `CF` tuples from all incoming links are available. It then evaluates the join-condition and writes tuples to its output places depending on the result of the evaluation: `CF` tuples are written to places *Start* and *Incoming Token Type* on positive, `DP` tokens on negative results. This is the behavior for `surpressJoinFailure="yes"`, whereas `surpressJoinFailure="no"` causes `DP` tuples to be written to those places instead. Additionally, a `joinFailure` fault is signaled to place *Failed* via tu_3. Join-condition evaluation may also lead to a `subLanguageExecutionFault` (tuple tu_2) if an error during expression evaluation occurs.

Transition t_2 is responsible to synchronize execution, i.e. to continue with control flow after the contained activity is finished. Since the contained activity may again be a `flow` that uses DPE, the original token resulting from evaluating the join-condition is preserved. That way we can avoid that Dead-Path-Elimination crosses the border of activities, e.g. if the `flow` in the contained activity results in a `DP` token in place *Ended*, this token is read by t_2. If however the join-condition evaluated to `true`, a `CF` token instead of the `DP` token must be written to place *Ended* of the link-target EWFN to avoid the DPE that originated in the `flow` that is the link target propagates to the `flow` where the link was defined.

`DP` token in place *Start* `DP` tokens propagate link-state and are used to synchronize a join, i.e. the join-condition is evaluated only after a control

flow token (either CF or DP) has arrived from each of the incoming links. DP tokens thus are forwarded by activities on their way through the EWFN graph, forcing activities to skip their business logic and to immediately forward the token to all directly succeeding activities.

5.3.9 ForEach

The forEach activity embeds a scope that is executed multiple times, either serially or all instances in parallel. The number of branches started may be computed at runtime and is controlled using expressions for startCounterValue and finalCounterValue. An optional completionCondition may be used to cause premature end of the loop or to forcefully terminate scope instances in the parallel case. A branches element further allows to define a completion condition that "at least n out of m" instances must be executed.

Figure 5.22 shows the EWFN pattern that implements the forEach activity, corresponding template and tuple definitions used are listed in Table 5.14. Each branch or iteration of the forEach activity is executed in a separate scope. The scopes are dynamically created depending on the number of iterations to be executed. In the following, we discuss the EWFN pattern based on two scenarios: arrival of a CF or a DP token in place *Start*.

Table 5.14: Template and Tuple Definitions for Figure 5.22

Symbol	Operation	Definition
tu_1	write	⟨fault:invalidExpressionValue⟩
tu_2	write	⟨fault:invalidBranchCondition⟩
tu_3	write	⟨counterName, counterValue⟩
tu_4	write	⟨scopeID⟩
tu_5	write	⟨initialize default message exchange⟩
tu_6	write	⟨fault:completionConditionFailure⟩

CF **token in place *Start*** A CF token in place *Start* enables transition t_1 that evaluates the expressions for startCounterValue and finalCounterValue and signals an invalidExpressionValue (tu_1)

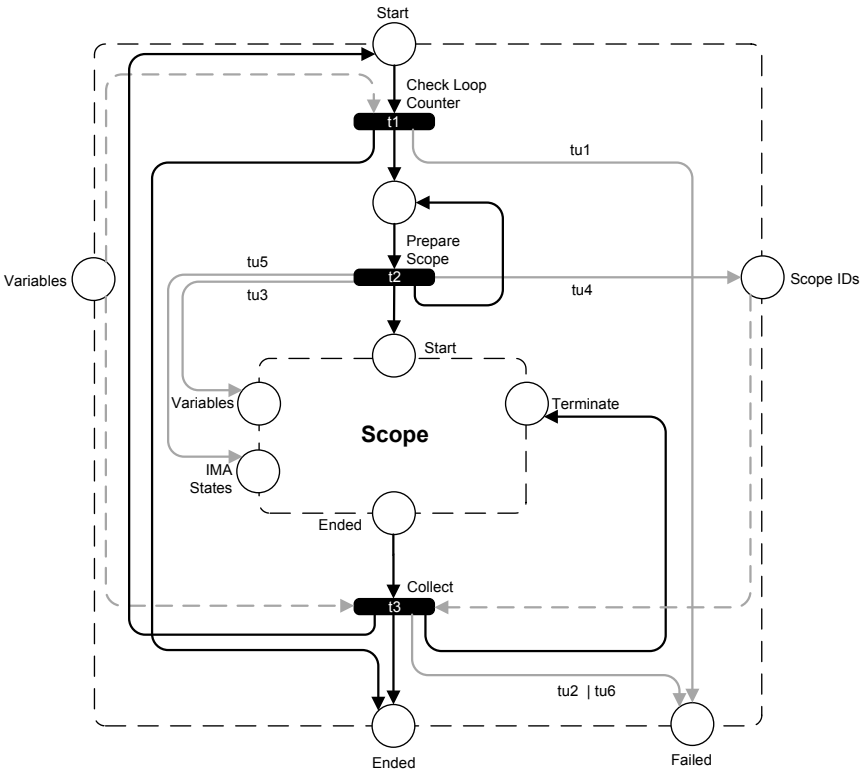


Figure 5.22: EWFN for `forEach`

fault if an error occurs during evaluation. If the `finalCounterValue` is greater or equal to the `startCounterValue`, the `scope` is not executed and a `CF` tuple is directly written to place *Ended*. Otherwise, transition t_2 initializes either one `scope` at a time (serial execution) or as many scopes as iterations are required (parallel execution, indicated by the self-activating arc leaving t_2 and writing directly to its input place). `scope` initialization (see Section 5.4 for details) includes adding the `counterVariable` (tu_3) to the *Variables* place of the `scope`, initializing a default message exchange (tu_5) via place *IMA States* and preparing IDs

to uniquely identify the `scope` instance (tu_4).

`scope` execution is governed by transition t_3, checking the `completionCondition` after each completion of the enclosed `scope` and terminating remaining scopes if their number is greater or equal to the number of scopes running (in the parallel case) or about to be run (in the sequential case). A `completionConditionFailure` (tu_6) is written to place *Failed* together with a `DP` tuple in place *Ended* if it can be determined that it will never evaluate to `true`. Transition t_3 is also responsible to evaluate the `branches` condition that may also lead to forced termination of running scopes. An error in branch condition evaluation is signaled by a `invalidBranchCondition` fault (tu_6), e.g. if its result is greater than the number of enclosed activities. If `successfulBranchesOnly="yes"`, transition t_3 counts only scopes that return a `CF` token on place `Ended`.

An error, indicated by a `DP` token in place *Ended* of the contained `scope` and a corresponding fault tuple in place *Failed* causes transition t_3 to end the `forEach` activity. It forwards the `DP` token to place *Ended* and does not start further iterations.

`DP` **token in place *Start*** Similar to the `while` activity, a `DP` token is directly forwarded to place *Ended*; link states within a contained `scope` do not have to be updated since links are not allowed to cross the enclosing `forEach` activity.

5.4 Scope

A `scope` defines the context for the activities it encloses, i.e. contained activities share access to `variables`, `partnerLinks`, `messageExchanges` and `correlationSets` defined within the given `scope`. It also may define additional behavior, e.g. through `eventHandlers`, `faultHandlers`, `compensationHandlers` and `terminationHandlers`. A `scope` and thus the context it provides can be nested hierarchically, leading to a tree structure of contexts (see Section 5.4.3 and 5.4.5 for details) with the context provided

by the `process` element as root node. The `process` element itself essentially denotes a `scope`, with the exception that it does not allow compensation and termination handlers to be defined and also cannot be declared to be *isolated* (see 5.4.7).

5.4.1 Scope Pattern

The `scope` activity embeds a *primary* activity e.g. it may contain a `flow` that itself contains a number of activities which are structured, basic or a `scope` again. Activating a `scope` includes activating all event handlers it defines together with the contained activity. Upon termination, all IMAs defined within the `scope` e.g. referenced through a `messageExchange`, must be completed. Otherwise, the IMA is called *orphaned* and a corresponding `missingReply` fault must be thrown.

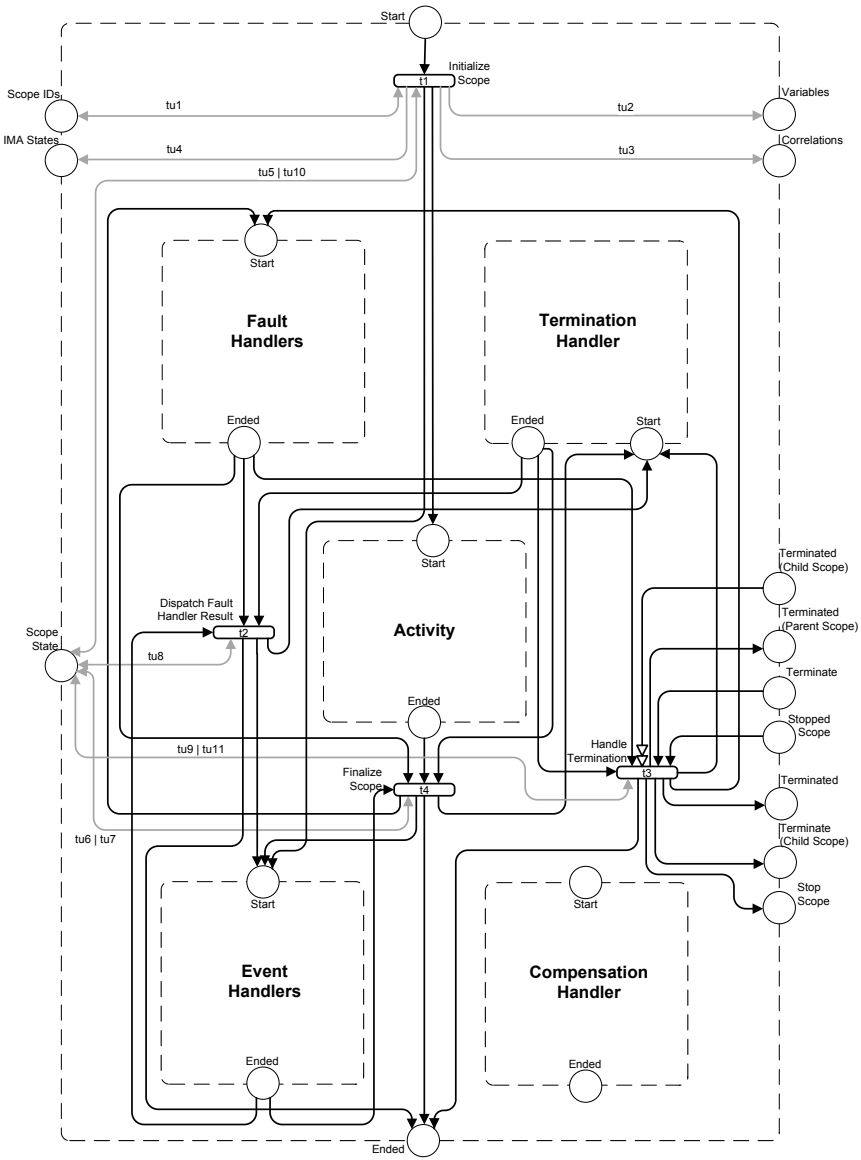


Figure 5.23: EWFN for `scope`

A `scope` may terminate for three different reasons:

Regular termination If the contained activity terminates without a fault and no orphaned IMAs are present, the `scope` waits until active event handlers are completed, then installs the compensation handler and gives control flow back to the activity that follows.

Termination caused by internal fault If an enclosed activity signals a fault, all contained activities as well as running event handlers are terminated (see Section 5.4.4). Then, a corresponding fault handler is notified to process the fault.

External termination An external condition (e.g. a terminating parent `scope` or a completion condition of a parallel `forEach`) signals the `scope` to terminate. This causes all event handlers and running activities enclosed by the `scope` to be terminated as well.

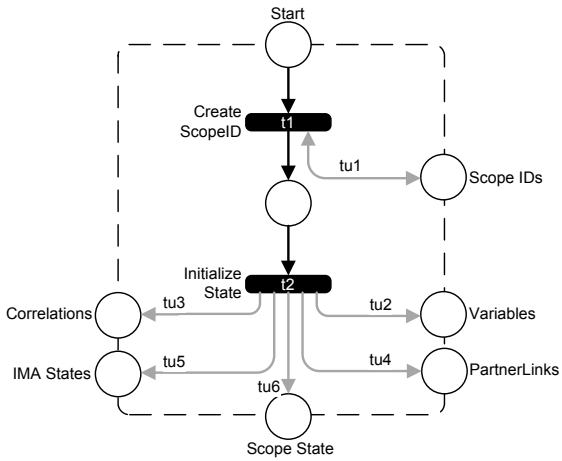


Figure 5.24: EWFN for the “Initialize-Scope” Composite Transition

Figure 5.23 shows the EWFN pattern that implements the `scope` activity, corresponding template and tuple definitions are given in Table 5.14. The

pattern consists of five different "pattern composition interfaces", one for each type of handler that may be declared within the `scope` and one for the primary activity, i.e. the process logic enclosed by the `scope`. Since it is a rather complex construct, four transitions – all of them being composite – are used to implement its semantics. Scopes can be nested arbitrarily and need to communicate between each other, e.g. for the purpose of termination. This is implemented by shared places that describe the name and the pattern they are equal to in parentheses. For instance, the place *Terminate (Child Scope)* is equal to place *Terminate* of the child `scope` which needs to be terminated

Note that the compensation handler is not directly connected to any transition. This is because a compensation handler EWFN is embedded into the EWFN of the activities `compensate` or `compensateScope` since its execution is controlled by these activities.

A `scope` behaves differently, depending on its current state, which is reflected in tuple form in place *Scope State*. This place is also used to store the states and timestamps of start and completion of all direct child scopes in case default compensation needs to be carried out (cf. Section 5.4.5). The following states – consistent with the `scope` state diagram presented in the BPEL event model [KKS+06, Ste08] – are used:

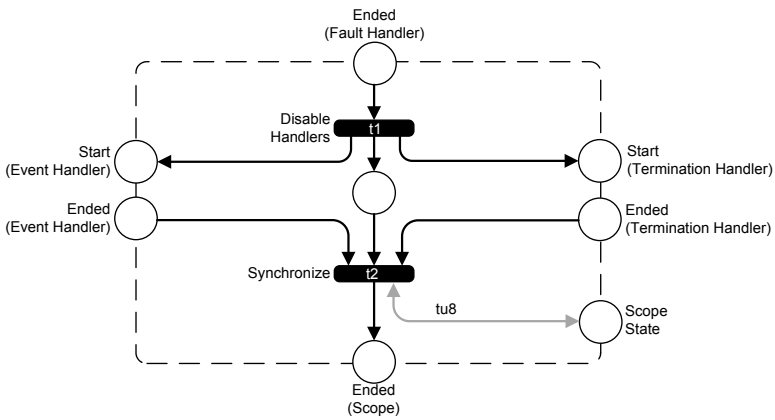


Figure 5.25: EWFN for the "Dispatch Fault Handler" Composite Transition

not_running This is the default state a `scope` is in if control flow did not arrive yet. A `scope` in this state cannot be forcefully (i.e. requested by a parent `scope`) terminated. This is the initial state of a `scope`.

running A `scope` in this state is running, i.e. the primary activity is running and the event handler is active. Forced termination may be applied.

dead A `scope` is in state "dead" if it will not be executed due to the effects of DPE. A `scope` in this state cannot be forcefully terminated. This is a final state.

failed A `scope` in state "failed" cannot be forcefully terminated. Also, event and compensation handlers are disabled at this point and the termination handler is disabled. This is a final state.

ended This state indicates successful completion of a `scope`. Forced termination cannot happen at this point. The event and termination handlers are disabled. However, a compensation handler may still run.

compensated A `scope` in this state was compensated by a compensation handler. Neither termination, event, fault or compensation handlers, nor the primary activity can run. This is a final state.

terminating This state indicates a running forced termination requested by a parent `scope`. This state prevents a fault handler to start terminating the currently terminating `scope`.

terminated This state indicates a completed termination of the `scope`. The parent `scope` uses this state to be informed when forced termination is completed.

In the following, we discuss the `scope` EWFN pattern based on two scenarios: arrival of a `CF` or a `DP` token in place *Start*.

`CF` **token in place *Start*** A `CF` token in place *Start* enables composite transition t_1 that is responsible for initializing the `scope`. The semantics of t_1 is detailed in Figure 5.24. It consists of two transitions that initialize

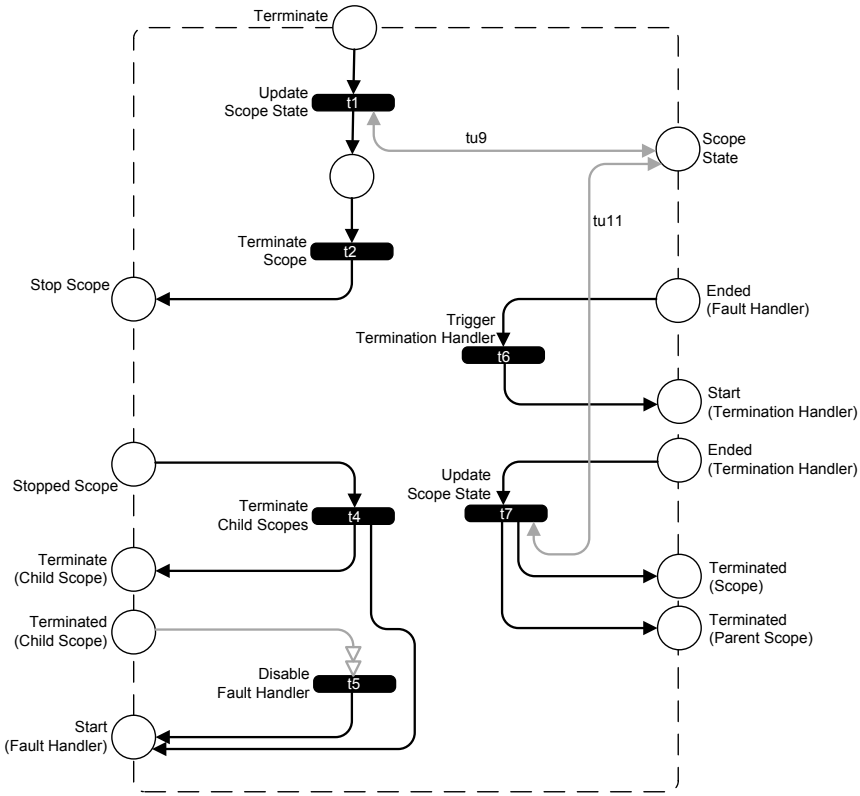


Figure 5.26: EWFN for the “Handle Termination” Composite Transition

the `scope`. Transition t_1 ensures that, before any tuple is created, a new `scope` ID is generated and written to place *ScopeIDs*. From now on, the control flow tuple also carries `scope` nesting information (see Section 5.4.2 for further information). Transition t_2 then initializes the remaining places by writing the tuples tu_2 , tu_3, tu_4 and tu_5 to their respective places, i.e. it initializes the places *Variables*, *CorrelationSets*, *IMA States* and *Scope State*. Control flow is then given to the primary activity by writing a `CF` tuple to its *Start* place.

Table 5.15: Template and Tuple Definitions for Figure 5.23, 5.24, 5.25, 5.26 and 5.27

Symbol	Operation	Definition
tu_1	write	⟨scopeID⟩
tu_2	write	⟨scopeID, variableName, initialValue⟩
tu_3	write	⟨partnerLink, operation, correlationset, "open"⟩
tu_4	write	⟨partnerLink, operation, messageExchange, "open"⟩
tu_5	write	⟨scopeID, "running"⟩
tu_6	write	⟨scopeID, "ended"⟩
tu_7	write	⟨scopeID, "compensated"⟩
tu_8	write	⟨scopeID, "failed"⟩
tu_9	write	⟨scopeID, "terminating"⟩
tu_{10}	write	⟨scopeID, "dead"⟩
tu_{11}	write	⟨scopeID, "terminated"⟩

Successful completion of the primary activity is indicated by a `CF` tuple on its *Ended* place, which is then processed by transition t_4 in order to finalize the `scope`. t_4 again is a composite transition which is detailed in Figure 5.27. Here, Transition t_1 updates the link states of the fault, termination and event handlers by writing `DP` tokens on the *Start* place of each of them. Transition t_2 synchronizes the `DP` tokens from their respective *Ended* place and updates the `scope` state to "ended" with tuple tu_6 so that the parent `scope` for instance cannot forcefully terminate this `scope`.

A fault signaled by the primary activity causes the fault handler of the `scope` to take over control (see Section 5.4.3 for details). Completion of the fault handler is communicated by a control flow token on its *Ended* place. A `CF` token indicates that no fault occurred during execution of the fault handler, whereas a `DP` token and a respective fault tuple in place *Failed* means that a fault occurred during execution. In both cases, transition t_2 (detailed in Figure 5.25) disables event and termination handlers by writing `DP` tokens to their *Start* places, waits until these

tokens show up again on their *Ended* places and then updates the `scope` state to "failed" via tu_8. In the case of a `CF` token, a `CF` token is also written to place *Ended* of the `scope` so that the link state can now be evaluated if the `scope` was the source activity of a link in an enclosing `flow`. A `DP` token together with a fault tuple in place *Failed* indicates that a fault occurred during execution of the fault handler. Consequently, the fault is propagated to the parent `scope` (see discussion of the fault handler in Section 5.4.3) and a `DP` token is written to place *Ended* of the `scope`.

A third case that may happen is a forced termination through a parent `scope` activity. In this case, the running primary activity and all running event handlers must be terminated. Subsequently, the termination handler of the `scope` is executed. To signal forced termination, the parent `scope` writes a `CF` token to the *Terminate* place of the `scope`. Composite transition t_3, detailed in Figure 5.26, is then responsible to coordinate the required actions. Its first action is to update the `scope` state to "terminating" via transition t_1. This prevents the fault handler from being run, e.g. if a fault occurs in the primary activity during termination. The next step, initiated by t_2 is to halt all activities in the `scope`, then recursively in all child scopes. The actual halting of execution is implemented by functionality provided in the tuplespace middleware triggered by a `CF` token in place *Stop Scope*. The tuplespace middleware writes a `CF` token to place *Stopped Scope* when it has finished its task. Next, t_4 triggers termination of all child scopes by writing a `CF` tuple on the *Terminate* places of all child scopes. A `sync` arc over the *Terminated* places of the child scopes ensures that transition t_5 is only enabled when all child scopes are terminated. If no child scopes are present, transition t_4 directly disables the fault handler and skips transition t_5. t_5 disables the fault handler of the `scope` by writing a `DP` token on its *Start* place. The `DP` token will be forwarded by all activities within the fault handler and eventually end up in its *Ended* place, enabling transition t_6 that writes a `CF` token to the *Start* place of the termination handler. Either a `CF` or

a DP token will be on the *Ended* place of the termination handler after execution. Either token enables transition t_7 which updates the scope state to "terminated" and signals completion of scope termination to the parent scope with a token in place *Terminated*. Since a fault that occurred in a termination handler is not propagated to the parent scope, both, a CF and a DP token enable transition t_7.

DP **token in place *Start*** A DP token is directly forwarded to the primary activity of the scope and eventually ends up at its *Ended* place. It is then forwarded to the *Ended* place of the scope by transition t_4.

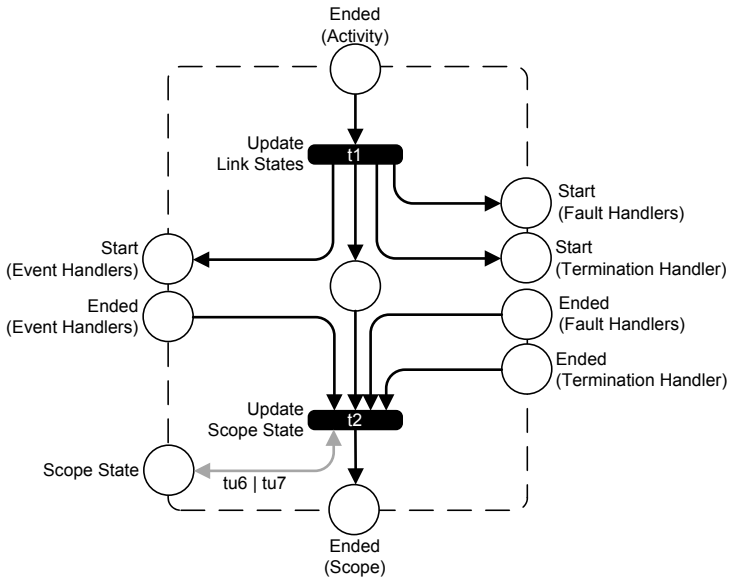


Figure 5.27: EWFN for the "Finalize Scope" Composite Transition

Note that there is no dedicated EWFN for the process activity since it is very similar to a scope. The difference is that it cannot contain a termination or a fault handler, respective places and arcs are therefore removed. The removed places in detail are: *Terminate*, *Terminated* and *Terminated (Parent Scope)*).

Also, the isolation attribute is not supported, i.e. the solution using a mutex as discussed in Section 5.4.7 does not apply.

5.4.2 Dynamic Scopes and Scope Nesting

BPEL scopes can be dynamically created (e.g. the number of scopes created is being calculated at runtime in a `forEach` activity) and nested to arbitrary depth, requiring complex logic to distinguish the different contexts created and to enforce the visibility rules for `variables`, `partnerLinks`, `messageExchanges` and `correlationSets`. Activities accessing context such as a variable for instance need to be aware of the current nesting of the scopes they reside in. Furthermore, dynamically created scopes need a way to distinguish their context from other scopes, especially when on the same level in the `scope` hierarchy.

We implement this by a scopeID that each variable carries, and the *context tuple*, that is part of every control flow tuple. The context tuple tells the currently active activity the identifiers and the structure of scopes it has access to. If e.g. a variable it accesses is not part of a `scope` identified by the contents of the context tuple, this variable is out of scope for the activity; it must not access its values. The context tuple itself is modified at each entry and exit of a `scope`: entering a `scope` causes the scopeID of said `scope` to be added as last element of the tuple, leaving a `scope` causes the current scopeID to be removed. Note that on `scope` exit, the current scopeID will always be the last element of the `scope` context tuple. ScopeIDs in the context tuple therefore not only denote currently visible scopes at the current point of control flow, but also the current `scope` nesting level. An activity that receives a `CF` tuple therefore has all required information at hand to enforce the visibility rules implied by the nesting of scopes in the process model.

Figure 5.28 shows an example of this concept. It is a simplified EWFN of a process with an activity enclosed by a `scope` that is the primary activity of a `forEach`. It is a mix of a static `scope` (the `process` element) and dynamically created scopes by the `forEach` activity. Consider for example a running process instance with scopeID $= 1$ for the `scope` defined by the `process` element and

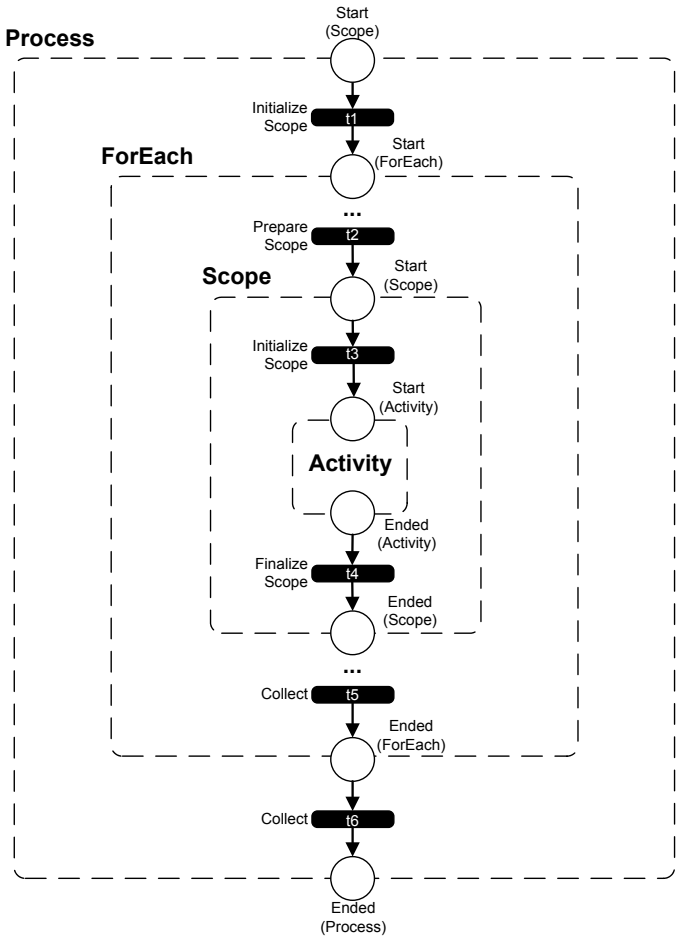


Figure 5.28: Nested `scope` Example

three scopes created by the `forEach` construct, identified by scopeIDs 11, 22 and 33 respectively. The `CF` tuple created by transition t_1 therefore is defined as

$$\texttt{CF} = \langle \text{PID}, \text{IID}, \text{“CF”}, \langle 1 \rangle \rangle$$

The CF tuples created by t_3 are:

$$\begin{aligned} \texttt{CF} &= \langle \text{PID}, \text{IID}, \text{“CF”}, \langle 1, 11 \rangle \rangle \\ \texttt{CF} &= \langle \text{PID}, \text{IID}, \text{“CF”}, \langle 1, 22 \rangle \rangle \\ \texttt{CF} &= \langle \text{PID}, \text{IID}, \text{“CF”}, \langle 1, 33 \rangle \rangle \end{aligned}$$

Variables initialized in the respective scopes are defined as variable

$$A = \langle \text{PID}, \text{IID}, \langle 1 \rangle, \text{value} \rangle$$

which is defined at the process level, and the variables

$$\begin{aligned} B_{11} &= \langle \text{PID}, \text{IID}, \langle 1, 11 \rangle, \text{value} \rangle \\ B_{22} &= \langle \text{PID}, \text{IID}, \langle 1, 22 \rangle, \text{value} \rangle \\ B_{33} &= \langle \text{PID}, \text{IID}, \langle 1, 33 \rangle, \text{value} \rangle \end{aligned}$$

which are defined by the three dynamic scopes created by the `forEach`.

As a result, an activity defined at the process level that receives a CF token with context tuple $\langle 1 \rangle$ is restricted to access only variable A, whereas an activity in the `scope` with ID 33 can access both variables, A and B_{33}, according to the context tuple $\langle 1, 33 \rangle$ it carries as part of the CF token.

Implementation details of this concept and more information about the concrete tuple structure used are available in [Wut10].

5.4.3 Fault Handler

Fault handling in BPEL exists to provide an alternative execution path if normal execution failed. Alternative execution paths are defined by *fault handlers* which operate at `scope` level. A fault that occurs during execution of the scope's primary activity causes normal execution to stop and a fault handler – defined to be responsible for the fault thrown – to take over.

Faults itself can be raised (i) by Web service interactions (e.g. a WSDL fault as response of an in/out operation), (ii) by the workflow engine during execution as indicated in the specification (a.k.a. BPEL standard fault), (iii) explicitly by the process modeler using the `throw` activity. If no explicit `catchAll` is

specified at a given `scope`, a default fault handler for all uncaught faults is installed that invokes the compensation handler and then re-throws the fault to the parent `scope`. If a fault occurs during execution of a fault handler, the fault handler is terminated and the fault is forwarded to the parent `scope`.

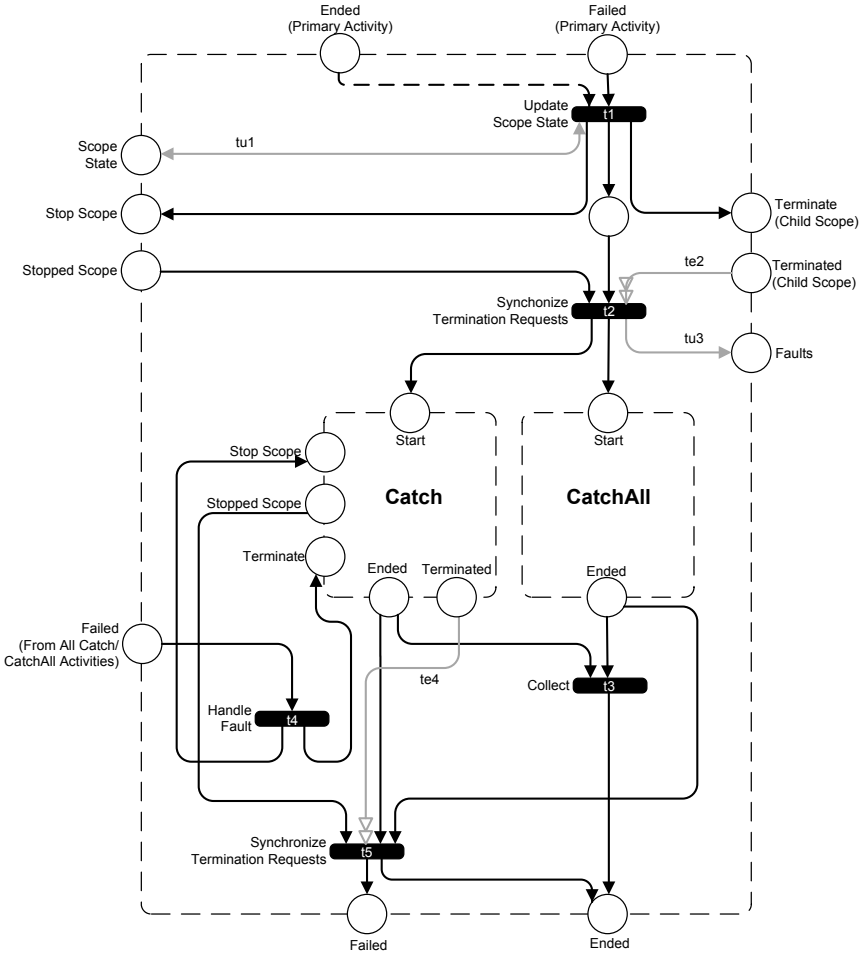


Figure 5.29: EWFN for Fault Handlers

Figure 5.29 shows the EWFN pattern for the fault handler, corresponding template and tuple definitions are listed in Table 5.16. As a fault handler may contain an arbitrary number of catch branches, each branch is modeled as a separate pattern. To exemplify this approach, Figure 5.29 contains two branches, one `catch` branch and one `catchAll` branch. Fault handling generally requires a lot of interaction with other constructs in the overall EWFN graph, e.g. to request termination of a child `scope` or to receive a signal from the executing middleware that a `scope` has terminated. Consequently, interface places in this pattern are annotated not only with a name, but also with the pattern (in parentheses) where this place occurs. Place *Terminate (Child Scope)* for instance is equal to all places named *Terminate* in all direct child scopes of a given `scope`.

In the following, we discuss the fault handler EWFN based on four scenarios: (i) a fault was raised by the scope's primary activity, (ii) a fault occurred within the fault handler itself, (iii) the fault is re-thrown to the parent `scope` by a `rethrow` activity in the fault handler, and (iv) a `DP` token arrives at the `scope`.

Table 5.16: Template and Tuple Definitions for Figure 5.29

Symbol	Operation	Definition
tu_1	write	⟨scopeID, "failed"⟩
te_2	sync	⟨`CF`, `CF`, ...⟩
tu_3	write	⟨faultName, faultValue⟩
te_4	sync	⟨`CF`, `CF`, ...⟩

Fault raised by the primary activity A fault raised by the primary activity causes two tokens to be produced: (i) a `DP` tuple is passed on along the control flow edges in order to update link states to `false`. A `DP` tuple therefore eventually ends up on place *Ended* of the activity that faulted. At the same time, a fault token such as tu_1 defined in Table 5.14 is written to place *Failed* of the failing activity, which is equal to place *Failed* of the enclosing `scope`. Note that this is a general concept: place *Failed* of all activities nested within a `scope` is always equal to place *Failed* of their directly enclosing `scope`. The fault token together with

the DP token on place *Ended* in this scope enable transition t_1 of the fault handler pattern depicted in Figure 5.29.

Transition t_1 is responsible for two tasks: it updates the scope state using tuple tu_1 and starts terminating all child scopes by writing a CF tuple to place *Terminate (Child Scope)*. However, these steps are only executed if the current state of the scope is "running". If the scope is in state "terminating" or "failed", transition t_1 ignores the incoming fault since the scope is either currently terminating or has already processed a fault.

In [Wut10], transition t_1 of this EWFN marks a very important spot. It is the transition from normal, forward control flow to "exceptional" control flow, i.e. control flow triggered by a fault. This differentiation is important to implement the functionality that removes normal forward control flow tokens, but *not* exceptional control flow tokens; fault handling still must be able to proceed. Only control flow tokens in the primary activity need to be removed, not the ones governing fault handling.

To terminate a given scope and all of its child scopes, place *Terminate (Child Scope)* and functionality provided by the executing middleware (see Section 5.4.3.1) is used. Place *Terminate (Child Scope)* is equal to all *Terminate* places of immediate child scopes, i.e. if the given scope has two child scopes, place *Terminate (Child Scope)* is equal to place *Terminate* of both child scopes. Also, the executing middleware is informed to terminate the current scope by writing a CF tuple to place *Stop Scope*.

Transition t_2 synchronizes the termination requests by issuing a sync operation with the number of templates equal to the number of child scopes, waiting for a CF tuple from each. It also waits for the DP tuple generated by the failing activity that is propagated to place *Ended* of the primary activity and for a CF tuple by the middleware to confirm termination of the scope. Once all templates are satisfied and transition t_2 fires, it is ensured that all child scopes are terminated, their termination handlers were executed and all links within the failing scope plus links leaving that scope are updated. No further faults may occur within this

`scope`. Starting from this point, an appropriate fault handler – depending on the fault that was raised – takes over control. For this to happen, t_2 produces a `CF` token in place *Start* of the chosen handler, and a `DP` token in place *Start* of the remaining fault handlers. At the same time, t_2 also persists the fault as tu_3 in place *Faults* allowing it to be possibly re-thrown later.

Once the fault handler completed successfully (indicated by a `CF` token in place *Ended*) and the `DP` tokens of the remaining handlers are also in their *Ended* places, transition t_3 fires and finally writes a `CF` tuple to place *Ended* of the fault handler pattern to signal successful processing of the fault. Transition t_3 synchronizes control flow within the fault handler, and is enabled if a `CF` or a `DP` token is available on each of the incoming arcs. When this point is reached, the primary activity, event, compensation and termination handlers of the `scope` are disabled or terminated and a fault handler was run, i.e. a final state of the `scope` has been reached.

Please note that transition t_4 and t_5 will never be enabled in this scenario: t_4 is enabled by a fault tuple in place *Failed (Catch / CatchAll)*, i.e. if a fault occurs in any of the fault handler branches; t_5 shares its enablement condition with transition t_3, however, additional tuples must be present, i.e. a `CF` token in place *Stopped Scope* and *Terminated (Fault Handler)*.

Fault occurred within a catch branch If a fault occurs during execution of the catch branch, a `DP` token ends up in place *Ended* of the failing branch. Additionally, a fault description token is written to place *Failed* of the `catch` branch. Place *Failed (Catch / CatchAll)* in the fault handler pattern is equal to all *Failed* places of the `catch` / `catchAll` branches. A fault description tuple in this place therefore indicates that a fault occurred during execution of the fault handler. Transition t_4 in this case starts terminating the fault handler by writing a `CF` token to its *Terminate* and *Stop Scope* places. Note that for simplicity, t_4 in this EWFN only terminates the `catch` branch. Although not explicitly drawn, the `catchAll` branch is terminated in the same way. Transition t_5 then

synchronizes the termination request by waiting for a `CF` tuple from place *Stopped Scope*, termination acknowledgments (in the form of `CF` tuples via place *Terminated*) from all child scopes of the fault handler branch that faulted, a `DP` token from the failing branch and `DP` tokens from all other branches. Transition t_5 then forwards the fault to the parent `scope` by writing a `DP` token to place *Ended* and a fault description tuple to place *Failed*.

Fault is being re-thrown by fault handler A fault handled by a fault handler may also be re-thrown to the parent `scope` to be handled there. Two cases for this "fault forwarding" can be distinguished: (i) an explicit re-throw activity may be used as the last activity of a catch branch, or (ii) the fault occurred during execution of the catch branch itself and is therefore implicitly forwarded to the parent `scope`. In the case where no fault handler is specified in the `scope`, a default fault handler, consisting of a `compensate` and a `rethrow` activity forwards the fault to the parent `scope`.

The EWFN for the `rethrow` activity is presented in Figure 5.30. This activity may only be used within a catch branch of a fault handler. To forward the fault it writes a `DP` token to its place *Ended*, which at the same time is place *Ended* of the fault handler since `rethrow` must be the last activity in the catch branch. The `DP` token is then forwarded to place *Ended* of the parent `scope` and, together with the fault description token written to place *Failed* of the parent `scope`, enables the parent scope's fault handler.

`DP` **token arrives in place *Start*** A `DP` token is forwarded to place *Ended* through each transition on the way. A `DP` token on place *Start* therefore always results in an `DP` token in place *Ended*. Note that this is similar to the cases of successful execution of the scope's primary activity (i.e. no fault occurred during execution) or forced termination: both also result in a `DP` token in place *Start* of the fault handler.

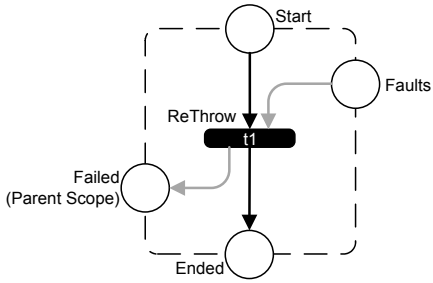


Figure 5.30: EWFN for `rethrow`

5.4.3.1 Scope Termination

As discussed in Section 5.4 it is required to be able to immediately stop – i.e. *terminate* – execution of a given `scope`. More specifically, terminating a `scope` refers to stopping control flow in the primary activity and possibly running event handlers *and* interrupting all running activities such as a blocking `receive` or a `wait` contained in the scope's primary activity.

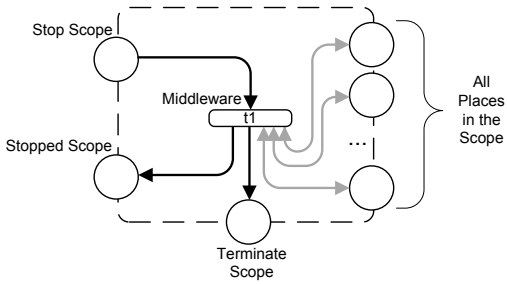


Figure 5.31: EWFN to Terminate a `scope`

Note that these requirements match with those for a scope-local version of the *exit* activity (cf. Section 5.2.9). `scope` termination is implemented by the EWFN in Figure 5.31: upon reception of a `CF` token in place *Stop Scope*, composite transition t_1: (i) informs all blocking activities of the `scope` to

immediately stop execution by writing a tuple in place *Terminate Scope*, and (ii) changes all `CF` tokens in the `scope` to `DP` tokens by issuing `update` operations on all places. As a consequence, all link states (see Section 5.3.6) of the `scope` are updated to a correct value; a deadlock caused by a join that waits for a link state to be set by DPE is prevented. Composite transition t_1 is implemented by the EWFN presented in Figure 5.13. Similar to the Process Termination EWFN (cf. Section 5.2.9), this EWFN is "virtual", it is implemented by the underlying execution middleware rather than by transitions (tuplespace clients). Details about its implementation are documented in [Wut10].

5.4.4 Termination Handler

A termination handler allows to specify user-defined behavior for the case of forced termination of a `scope`. By default – i.e. no user defined handler is specified – the termination handler compensates the terminated `scope`. A termination handler is only called when the respective `scope` was in state "running"; it will be ignored in all other cases, e.g. if the `scope` is in state "failed" or "terminating" the termination handler will not be executed.

A termination handler always runs before a fault handler is called, it can therefore be used to prepare work for the fault handler or to simply inform users that termination has occurred in the process. A fault that occurred within the termination handler is not propagated to the parent `scope`, however, the termination handler itself is terminated.

Figure 5.32 shows the EWFN pattern for the termination handler, corresponding template and tuple definitions are listed in Table 5.17. The termination handler pattern consists of a primary activity representing custom termination handler logic and three transitions that govern its execution, i.e. finalize normal execution, handle faults and synchronize termination. In the following, three situations are discussed: (i) normal execution of the termination handler, (ii) a fault occurring during execution of the termination handler and (iii) arrival of a `DP` token on place *Start* of the termination handler.

`CF` **token in place *Start*** As the places *Start* of the termination handler pattern overlaps with place *Start* of the handler's primary activity, a `CF` token

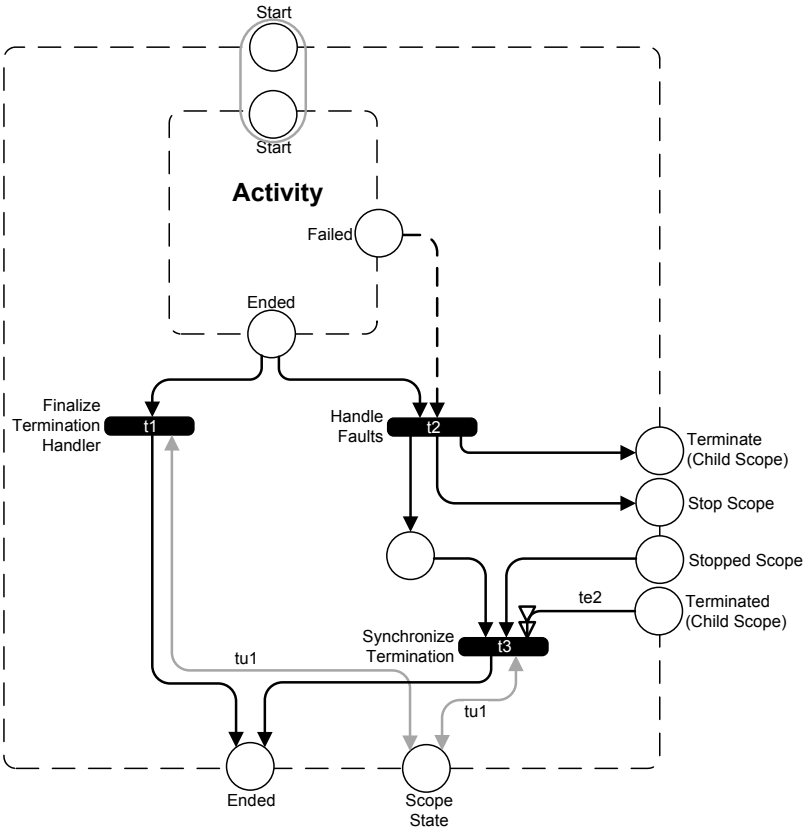


Figure 5.32: EWFN for Termination Handlers

(written by transition t_2 in the `scope` EWFN, see Figure 5.23) directly enables the primary activity of the termination handler. Upon its successful completion, a `CF` token is available on its place *Ended*, enabling transition t_1 that then finalizes the termination handler by updating the `scope` state using tuple tu_1 and writing a `CF` token to place *Ended* of the pattern. This ends normal processing of the termination handler.

Fault occurs during execution of the primary activity A fault that occurred

Table 5.17: Template and Tuple Definitions for Figure 5.32

Symbol	Operation	Definition
tu_1	write	⟨scopeID, "terminated"⟩
te_2	sync	⟨CF, CF, ...⟩

in the primary activity is indicated by a fault description tuple available in place *Failed* and a DP token in place *Ended*; both together enable transition t_2 to handle the fault. The actions triggered by this transition are three-fold: (i) termination of all child scopes of the handler (if existing) is requested by writing a CF token into place *Terminate (Child Scope)*, (ii) scope termination as described in Section 5.4.3.1 is triggered by a CF token in place *Stop Scope* and control flow is passed on to transition t_3. This transition synchronizes the termination requests send out earlier by waiting for a CF token in place *Stopped Scope* indicating that termination is finished, and ensures that there are as many CF tokens in place *Terminated (Child Scope)* as there are child scopes in the primary activity by issuing a respective sync operation. After all tuples are available, transition t_3 updates the state of the enclosing scope using tu_1 and writes a DP tuple to place *Ended*.

DP **token in place *Start*** A DP token in place *Start* can occur for two reasons: (i) Dead-Path-Elimination is going on in a flow enclosing the scope that holds the termination handler or (ii) the scope executes normally and the termination handler is deactivated. In both cases, the tokens update states of links that have source activities declared within the termination handler. As usual, the tokens are forwarded by each activity to its succeeding activities and eventually end up in place *Ended*.

5.4.5 Compensation Handler

The purpose of a compensation handler is to reverse the work of a successfully completed scope. Note that this is in contrast to a fault handler, which

tries alternative execution paths when a fault occurred. A faulted `scope` cannot be compensated. A compensation handler is invoked by the activities (i) `compensate` to compensate *all* child scopes or (ii) `compensateScope` to compensate a specific child `scope`, both activities being defined in a fault, compensation or termination handler (FCT handler) of the directly enclosing `scope`. The `compensate` activity is part of the default FCT handler, i.e. if no custom handler is defined for a given `scope`, the default behavior is to compensate all child scopes in *default order* (see default compensation order (DCO) in [A$^+$07]). If an compensation handler that was already executed is invoked for a second time, it behaves as if an `empty` activity was invoked. On completion of normal `scope` execution, a snapshot of the `scope` state, including `variables`, `partnerLinks`, `messageExchanges` and `correlationSets` is taken so that this state can be accessed later during compensation. Snapshots for scopes enclosed by a repeatable activity, such as a `while` for instance, are taken separately for each iteration. Since `scope` context tuples contain ProcessID, InstanceID and ScopeID, taking a snapshot means not to remove these tuples after the `scope` has completed, making the last state available to the compensation handler.

During the execution of the compensation handler, data shared with the rest of the process, i.e. variables in a parent `scope`, may be modified by other parts of the process running in parallel. Similarly, shared data modified by the compensation handler may be accessed by other parts of the process. A compensation handler is treated as another concurrently running part of the process, no data access preference or isolation is applied. If synchronization or isolation is desired, it has to be modeled explicitly in the compensation handler, e.g. using an isolated `scope` (see 5.4.7).

Figure 5.33 shows the EWFN pattern for the `compensation handler`, corresponding template and tuple definitions are listed in Table 5.18. The EWFN consists of a single transition, t_1, that is responsible to check the state of the enclosing `scope` using te_1, i.e. to decide if the compensation handler should be executed or skipped in case of a repeated execution. A `scope` state other than "ended" (e.g. the `scope` has not finished execution normally or was compensated already) results in skipping the contained activity and a `CF` token

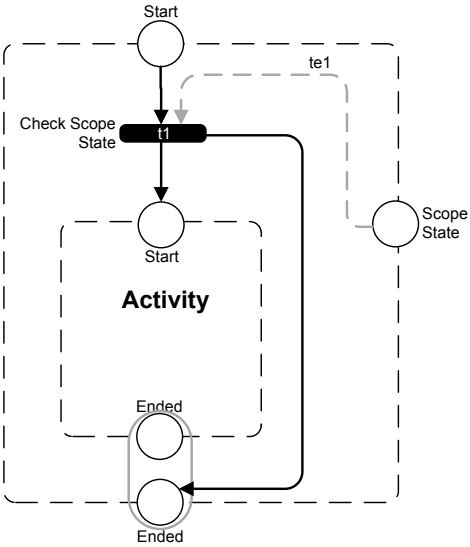


Figure 5.33: EWFN for Compensation Handlers

in place *Ended*. If the `scope` state was "ended" instead, transition t_1 forwards control flow to the embedded activity by writing a `CF` token to its *Start* place. Successful execution of the contained activity is indicated by a `CF` token in its place *Ended*, which is equal to place *Ended* of the compensation handler pattern.

Compensation handlers itself are executed and controlled by the so called compensation activities `compensate` and `compensateScope` (see Section 5.4.5.1). Their patterns also define the behavior of the compensation handler employed if a fault occurs during execution or a `DP` token is received. This is why the discussion of different execution scenarios for the compensation handler is also included in their section.

Table 5.18: Template and Tuple Definitions for Figure 5.33

Symbol	Operation	Definition
te_1	read	⟨scopeID, "ended"⟩

5.4.5.1 Compensate

The `compensate` activity instructs all compensation handlers of child scopes directly enclosed by the `scope` that hosts the FCT handler to be invoked in default order (cf. *default compensation order (DCO)* [A+07]).

The set of compensation handler instances of regularly finished scopes selected for compensation – in default order using `compensate`, or by directly selecting the `scope` to be compensated using `compensateScope` – is called the *compensation handler instance group* (CHIG). In the case of default compensation, compensation handler instances of that group are executed in default order, which is governed by two rules: (i) reverse order of control dependency, and (ii) prohibition of cycles in the *peer* `scope` *dependency relation* [A+07].

In this section, we discuss the EWFN implementation of `compensate` that invokes the CHIG in default compensation order. `compensateScope` in contrast invokes the compensation handler directly addressed by the `target` parameter of the activity.

Figure 5.34 shows the EWFN pattern for the `compensate` activity, corresponding template and tuple definitions are listed in Table 5.19. The pattern itself consist of three transitions: composite transition t_1 is responsible to execute the compensation handlers of the CHIG in DCO, transitions t_2 and t_3 are responsible for handling faults that occurred during compensation and for synchronizing the termination requests send to all child scopes of the compensation handler.

We discuss three situations that may occur during execution of `compensate`: (i) normal execution without a fault being raised, (ii) a fault occurs during execution of a compensation handler and (iii) arrival of a `DP` token on place *Start* of the `compensate` activity.

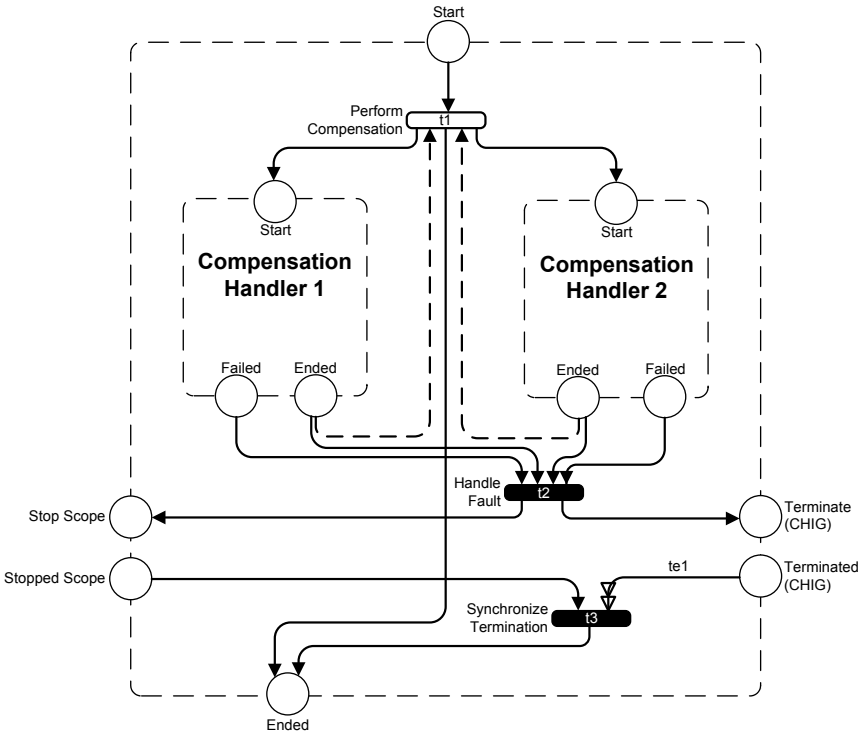


Figure 5.34: EWFN for `compensate`

`CF` **token in place *Start*** A `CF` token in place *Start* enables composite transition t_1 that invokes the compensation handler instances of the CHIG in default order. Note that this might also result in parallel execution of compensation handlers if implied by DCO. Transition t_1 is a composite

Table 5.19: Template and Tuple Definitions for Figure 5.34

Symbol	Operation	Definition
te_1	sync	$\langle \mathtt{CF}, \mathtt{CF}, \ldots \rangle$

transition that governs the execution of compensation handlers, it thus is connected to the *Start* and *Ended* places of each handler. The concrete implementation of composite transition t_1 is intentionally left open, since its internal behavior is discussed in [Wut10] in detail. Its functionality depends on recording the scopes where control flow tuples traveled through during execution to place *Scope State*. Using this mechanism, place *Scope State* provides information on the order of start and completion of all direct child scopes. With this information, the reverse order is calculated and the compensation handlers are invoked accordingly.

Successful execution of a compensation handler is indicated by a `CF` token in its place *Ended*. After the last handler in DCO gives back a `CF` token, transition t_1 writes a `CF` token to place `Ended` of the `compensate` pattern to signal that compensation has finished successfully.

Fault occurs during execution of the enclosed activity An uncaught fault – i.e. a fault that is propagated from a running compensation handler within the active CHIG – is signaled by a fault token in place *Failed* and a `DP` token in its *Ended* place. This enables transition t_2 that immediately terminates all compensation handlers of the active CHIG by writing a `CF` token to their *Terminate* place. Transition t_1 disables the remaining compensation handlers by writing `DP` tokens to their *Start* places.

While starting termination of the compensation handlers, transition t_2 stops their scopes by writing a `CF` token to their *Stop Scope* places, effectively using the `scope` termination EWFN as discussed in Section 5.4.3.1. Termination of the CHIG is then synchronized by transition t_3 that waits for `CF` tokens from all members of the CHIG to signal completed termination using a `sync` operation with the number of `CF` templates equal to the number of members in the CHIG. Transition t_3 also waits for a `CF` token on place *Stopped Scope* that indicates that `scope` termination is completed.

Note that a fault within a `scope` nested in a compensation handler is handled by the fault handler of said `scope` and may itself call the scope's compensation handler as modeled by the EWFN pattern for the `scope`

(Section 5.4). The fault then may be re-thrown to the parent `scope` and thus cause termination of the CHIG as described above. Already completed compensation handlers within the CHIG to be terminated are not further compensated.

`DP` **token in place *Start*** As links are not allowed to leave compensation handlers, it is not required to propagate `DP` tokens from a parent activity to the compensation handler. However, `DP` tokens are used *within* a compensation handler (if e.g. a `flow` is used) to indicate dead paths caused by join failures or a fault as usual. As a result, `DP` tokens do not cross the boundaries (neither inward nor outward) of a compensation handler EWFN pattern.

A `DP` token arriving at the `compensate` EWFN however causes it to be forwarded in the usual way since a compensate activity itself may be used as link source as well as a link target.

5.4.6 Event Handler

An event handler allows to raise new or react to events in parallel to normal processing logic of a `scope`. There are two different types of events: (i) inbound Web service operations and (ii) alarms raised after user-defined timeouts. Unlike FCT handlers, an event handler is part of the normal behavior of a `scope`. It is enabled when its associated `scope` is enabled and it is disabled when the `scope` ends. An event handler has exactly the same lifecycle as the associated `scope`. During lifetime, an event handler may be executed multiple times – each time an incoming message is detected by the engine and correlated to the current instance of the process. A fault in an event handler is forwarded to the fault handler of the associated scope; if the fault is not handled there or being re-thrown, it is forwarded to the parent scope's fault handler as usual. The child activity of an event handler is a `scope` again, i.e. all functionality defined for the event handler is defined within its associated `scope`.

Figure 5.35 shows the EWFN pattern for the event handler activity, corresponding template and tuple definitions are listed in Table 5.20. In this pattern,

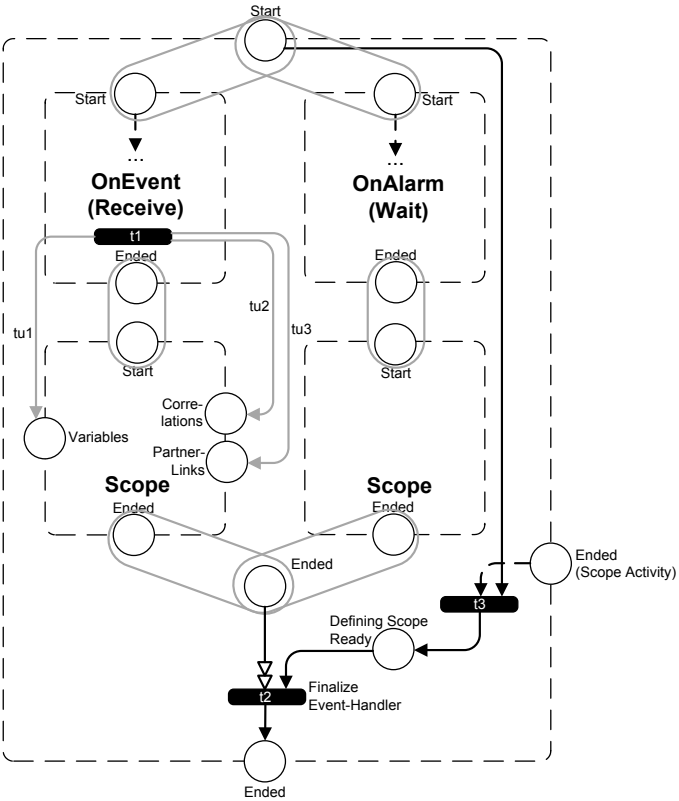


Figure 5.35: EWFN for Event Handlers

we reuse basic activities to provide support for the different types of events: a `receive` (see Section 5.2.1) is used for the functionality of inbound message events, the functionality of `wait` (Section 5.2.6) is used to implement alarms. Since an event handler cannot create a new process instance (its lifecycle starts with the activation of the defining `scope`), the non-instance-creating version of the `receive` pattern is used. The reused patterns are modified slightly to implement the semantics of the event handler: (i) the token in the shared *Start* place is *read* instead of taken, as a result, the `receive` and `wait`

activities (to implement the attribute `repeatEvery`) are enabled for the full duration of the lifetime of the defining `scope`. (ii) transition t_5 of the `receive` EWFN (Figure 5.2, called t_1 in the event handler figure) initializes `variables`, `partnerLinks` and `correlationSets` of the associated `scope` so that e.g. the variable used to hold the data received by the Web service operation can be directly used by activities within the `scope`.

In the following, we discuss three situations that may occur during execution of an event handler: (i) normal execution without a fault being raised, (ii) a fault occurs during execution of an event handler and (iii) arrival of a `DP` token in place *Start* of the event handler.

Table 5.20: Template and Tuple Definitions for Figure 5.35

Symbol	Operation	Definition
tu_1	write	⟨scopeID, variable⟩
tu_2	write	⟨scopeID, correlationSet⟩
tu_3	write	⟨scopeID, partnerLink⟩

`CF` token in place *Start* A `CF` token in place *Start* – written by the outgoing arc from transition t_1 in Figure 5.23 – enables all event handlers (inbound Web service communication and alarms) of the given `scope`. The token is not removed until transition t_2 fires to finalize the handlers. Therefore all handlers are active and may fire any time during the lifetime of the defining `scope`.

After the embedded activity (`receive` or `wait`) has finished, control flow is forwarded to the associated `scope` of the handler that contains the custom handler logic. In the case of an alarm, this happens automatically when a `CF` token is written to place *Ended* in the `wait` activity, since *Ended* of `wait` and *Start* of the associated `scope` are equal places. In the case of `receive`, a small modification of transition t_5 of Figure 5.2 is used that writes the result of the received data (`variables`, `partnerLinks` or `correlationSets`) to the associated `scope` so that the new data can be used directly. Similarly to the `wait` activity, a `CF` token in place *Ended*

of the embedded `receive` is equal to a `CF` token in place *Start* of the associated `scope` of the handler.

Normal finalization of the handler's associated `scope` is signaled by a `CF` token in place *Ended*, which is equal to the *Ended* places of all event handlers of a `scope`. Transition t_3 waits for the primary activity of the defining `scope` to be finished, then removes the `CF` tuple from place *Start* of the event handlers to disable event handling and puts a `CF` token in place *Defining Scope Ready*. Transition t_2 now issues a `sync` operation for as many `CF` tuples as event handlers were fired thus synchronizing the finalization of still running event handlers. Upon firing, transition t_3 puts a `CF` token in place *Ended* and signals finalization of event handling to the defining `scope`, which uses this information to itself finalize the `scope` in transition t_4 of the `scope` EWFN (Figure 5.23).

Fault occurs during execution of an event handler A fault that occurs during execution of the event handler, or more precisely the primary activity of the event handler's associated `scope`, is being handled by the fault handler of the associated `scope` using fault handlers as discussed in Section 5.4.3. If the fault is not caught or being re-thrown by the fault handler, it is forwarded to the fault handler of the parent `scope`.

`DP` token in place *Start* Since links are not allowed to cross event handlers, a `DP` token in place *Start* is directly written to place *Ended*.

5.4.7 Scope Isolation

A `scope` in BPEL may have its `isolated` attribute set to "yes" to control concurrent access to shared resources such as variables or partnerLinks. Isolated scopes ensure that the effects of conflicting data access operations by activities of different scopes are conceptually reordered in such a way that the effects are the same as if the conflicting scopes were executed sequentially. The BPEL specification [A+07] therefore describes the resulting effect on a `scope` to be similar to the *serializable* isolation level known from database transactions

[GR93]. Note that event handlers defined within an isolated `scope` are also part of the scope's isolation domain. Similarly, a compensation handler may be invoked such that it runs isolated although not defined in an isolated `scope`, e.g. when the invoker of the compensation handler runs within an isolated `scope`.

`scope` isolation in EWFNs is implemented through a *mutex* [Dij67] tuple shared by each isolation domain. All conflicting resources accessed by different isolated scopes are protected using a mutex per resource. An isolated `scope` that accesses one of these resources acquires the mutex tuple and releases it when all operations of the `scope` are finished, e.g. on `scope` finalization. Concurrently running isolated scopes that want to access the same resources are blocked until the mutex tuple is available again (and thus one of the scopes has finished). This is the same technique as employed in the instance-creating version of the `receive` EWFN pattern (see Figure 5.4 in Section 5.2.1) to detect conflicts in the concurrent initial set and serialize the creation of new process instances.

To implement "isolation infection" for child scopes or event handlers of an isolated `scope`, an *isolation `scope` ID* is added to the `CF` tuple communicated within an isolated `scope` that "infects" event and compensation handlers with the isolation domain. Further information and implementation details of this concept can be found in [Wut10].

5.5 Example: Simple BPEL Process as EWFN

Listing 5.36 shows the BPEL code of a simple synchronous process we use for demonstrating EWFN to BPEL transformation. The process consists of a `scope` (the top-level `process` element) that hosts a `sequence` of a `receive` (with the `createInstance` attribute set to "yes"), an `assign` and a `reply` activity. It receives a Web service message, copies the contents of the message to the reply variable and sends it back as a response to the Web service interaction.

Figure 5.37 shows the result of the translation of the presented BPEL process to EWFNs using the procedure discussed in Section 5.1.2. The figure shows how the `receive` pattern with `createInstance="yes"` overlaps with the `scope`

```
1  <bpel:process name="example_process">
2      <bpel:sequence name="main">
3          <bpel:receive name="receive" partnerLink="client"
4                      portType="tns:example_process"
5                      operation="process" variable="input"
6                      createInstance="yes"/>
7          <bpel:assign validate="no" name="Assign">
8              <bpel:copy>
9                  <bpel:from variable="input"></bpel:from>
10                 <bpel:to variable="output"></bpel:to>
11             </bpel:copy>
12         </bpel:assign>
13         <bpel:reply name="reply"
14                 partnerLink="client"
15                 portType="tns:example_process"
16                 operation="process"
17                 variable="output" />
18     </bpel:sequence>
19 </bpel:process>
```

Figure 5.36: BPEL Code of the Example Process

as indicated by transition t_6 in Figure 5.4. The upper part of the `receive` functionality (e.g. message reception, validation and correlation) is independent of a particular process instance and thus not part of the functionality carried out within the `scope`. Clearly, a `scope` cannot be instantiated at this point since a corresponding message has not been dispatched yet. Note that in this figure, the `scope` pattern itself was simplified by removing all functionality regarding fault, compensation and event handling since these are not used in the process. Also, for ease of readability, places with equal name denote equal places; the places are copied in order to avoid intersecting arcs. The arc labels used are the same from their original EWFN descriptions, e.g. tu_4 within the `scope` is described in Table 5.15 of Section 5.4.

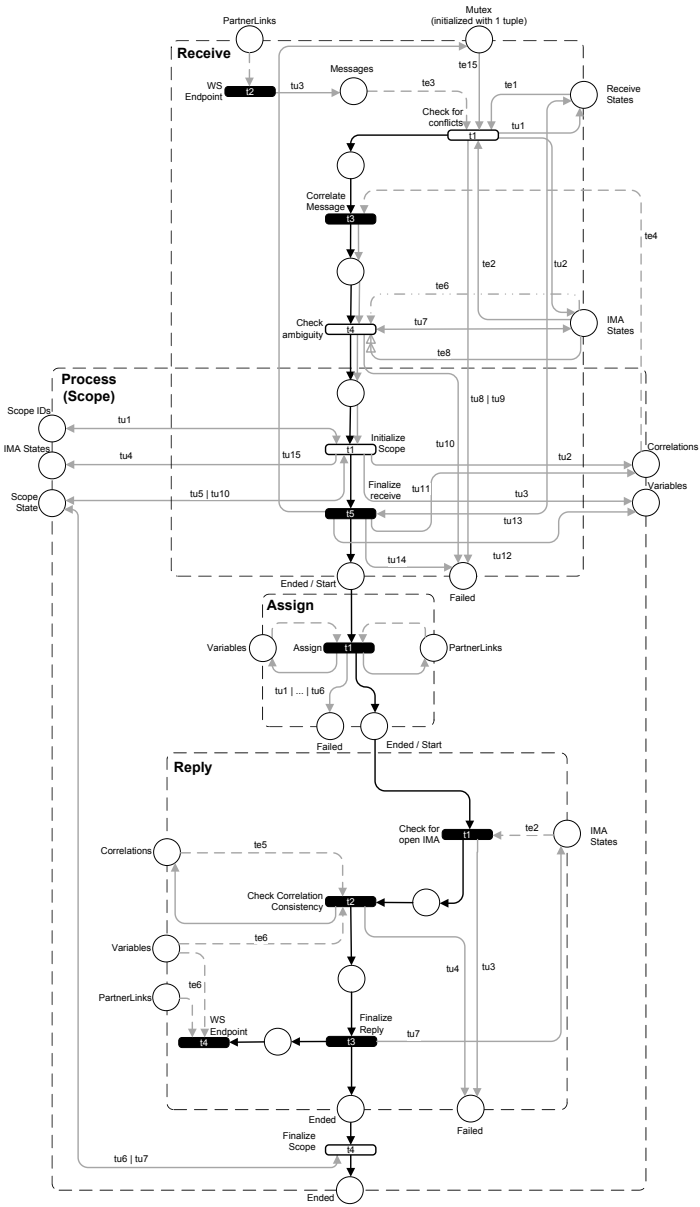


Figure 5.37: Simple BPEL Process as EWFN

5.6 Conclusion

In this chapter we discussed how the EWFN model defined in Chapter 4 can be used to execute WS-BPEL, a popular workflow definition language in use today. WS-BPEL is transformed to EWFNs by defining patterns that follow the same composition rules as their BPEL counterparts. In order to formalize BPEL in form of EWFN patterns, Section 5.1.1 presented modifications to set Σ, introducing new tuple types to be able to implement special features such as Dead-Path-Elimination or dynamically created scopes. Furthermore, a graphical notation to visualize EWFN patterns was presented as an extension to the basic EWFN notation introduced in Section 4.2.6.

Next, we discussed patterns for BPEL's basic and structured activities respectively. We introduced the semantics of each activity and played the "token game" to show how the EWFN pattern implements specific situations. We also discussed how terminating a process instance using the `exit` activity – which is especially hard due to the global semantics of this operation – is implemented using EWFNs.

This was followed by a discussion on how the `scope` activity, associated handlers, dynamic scopes and isolated scopes are implemented using EWFNs. Additionally, we have shown how fault handling and `scope` termination, two features of BPEL that are especially challenging to be implemented using Petri nets, are realized in our approach.

Finally, we provided an example of a small BPEL process and its equivalent EWFN composed from the individual patterns.

In the following chapter, we will present generic methods to execute Petri nets using tuplespaces and then refine these methods to be able to efficiently execute EWFNs.

CHAPTER 6

TUPLESPACE-BASED EXECUTION OF EWFNS

This chapter presents a detailed discussion on how EWFNs are executed on tuplespace middleware. We start by investigating generic methods of tuplespace-based Petri net execution and introduce strategies to map the structural information of a given Petri net into tuples (Section 6.1). The different mapping strategies are evaluated and tuple-Petri net representations for two widely used categories of Petri nets – Colored Petri Nets (CP-nets) and Place-Transition Nets (PT-nets) – are proposed.

As EWFNs are strongly based on Colored Petri Nets, the proposed generic CP-net mapping strategy also applies to EWFNs. Together with the BPEL 2.0 EWFN mapping presented in Chapter 5, a fundamentally new way of enacting business processes is created. Details of the resulting system, its unique properties compared to traditional workflow engines and the range of different deployments this new way of process execution allows, is discussed in Section 6.2.

Next, Section 6.3 focuses on algorithms that implement the individual oper-

ations of EWFNs. This includes implementations of basic tuplespace operations such as `read`, `write` and `take`, but also for implementations of the extension operations, e.g. `readall` or `update`.

Finally, Section 6.4 discusses the implementation of synchronization operations on tuplespaces that can e.g. be used for joining different paths of control flow. In particular, an algorithm for the `sync`-operation (cf. Section 5.1.1.1) is presented that solves this problem for the case where the requested tuples all reside in the same tuplespace. The developed algorithm is then used as the basis for the *sync-pattern*, that solves the problem of synchronized access to tuples residing in different, possibly distributed tuplespaces.

In this chapter, the term "execution model" describes the method how the workflow engine enacts a process, e.g. using token passing as in our case; the term "encoding model" describes the way how the structure of a Petri net is encoded in tuples.

6.1 A Generic Approach to Execute Petri Nets Using Tuplespaces

The term "Petri net" stands for a whole class of different Petri net types, such as Condition-Event Nets (CE-nets), Place-Transition Nets (PT-nets) or Colored Petri Nets (CP-nets). Common to all different forms is their basic structure, a bipartite graph with places and transitions as nodes. Individual Petri net types add certain elements or restrictions to the original structure, e.g. arc weights, tuple "colors", priorities or structural restrictions such as the ones for Free Choice Nets [DE95].

For the discussion that follows, we pick two prominent Petri net "dialects" from different levels of the Petri net classification [BdCdM92] and discuss how these can be executed using tuplespaces. The "high-level" Petri net dialect chosen is Colored Petri Nets [Jen92] (level 3 in the classification from Section 2.1.3), the other candidate is Place-Transition Nets (level 2 in the classification from Section 2.1.3). Since there exists a direct mapping from CP-nets to PT-nets, we first concentrate on the execution of PT-nets to find a generic solution – i.e. a method that allows to execute all kinds of Petri nets with a mapping to PT-nets. We find two different ways to execute PT-nets,

one of which turns out to be particularly applicable to PT-net execution while the other allows for the *direct* execution of CP-nets without the need for a transformation to PT-nets.

6.1.1 Analysis of Petri Net Tuple Encodings

Figure 6.1 shows a Place-Transition Net that models the basic steps of the well-known "Loan Approval" example process from the WS-BPEL 2.0 specification [A+07]. Note that the PT-net does not implement the BPEL semantics of the original process from the specification (such as DPE or the semantics of a `scope`), we rather just use it as a simple run-through example to show how a generic Petri net model can be executed using tuplespaces.

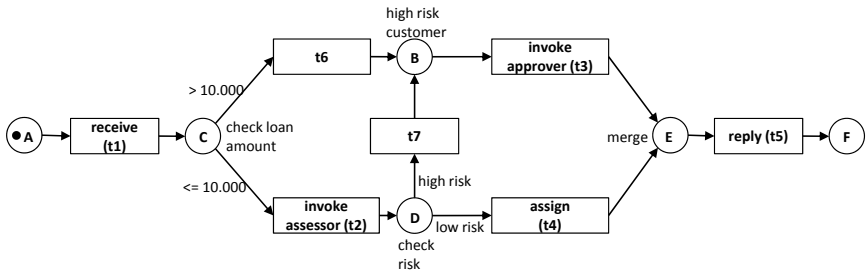


Figure 6.1: BPEL Loan Approval Process, Modeled as PT-net

The process is initiated by the reception of a customer request (i.e. a token in place A which enables transition t_1 called *receive*), invokes an external approver (e.g. a person that performs manual checks using transition t_3) if the loan requested is greater than 10.000 (upper branch), and returns the reply to the customer (transition t_5 called *reply*). If the loan requested is less than or equal to 10.000 (lower branch), a streamlined process is used invoking an internal, automated credit rating service to assess the risk (transition t_2 called *invoke assessor*); the manual approver is this case is only used if the risk was reported to be "high", else the loan is granted automatically and the customer is informed (t_5, *reply*). Note that the only functionality of transitions t_6 and t_7

is to forward tokens to place B from where the processing of "high risk" loans takes place. As usual with PT-nets, choices are modeled as a single place and competing transitions; in the example process e.g. through the places C and D.

Figure 6.2 shows the same process, but now modeled as CP-net instead of a PT-net. In this figure, we strictly follow the CP-net syntax proposed by [Jen92], i.e. each arc is annotated with a variable or an expression that evaluates to a variable; *empty* means that no token is communicated[1]. Each place is annotated with a color (*request*, *risk*, *approval*) denoting the type of tokens it may contain. Additionally, the multiset of tokens the initialization function for each place evaluates to is denoted as an underlined expression. Similarly, the current marking of a place is denoted by a small circle around the number in the multiset expression.

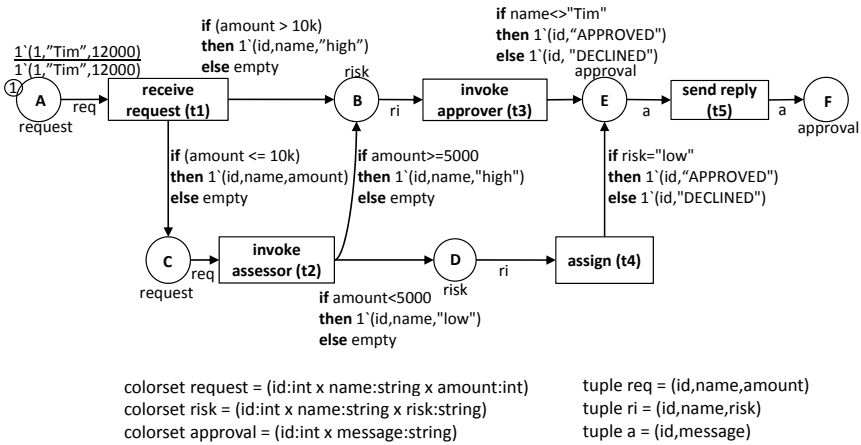


Figure 6.2: BPEL Loan Approval Process, Modeled as CP-net

Following the similarities of Petri nets and tuplespaces as discussed in Chapter 4, a straight-forward mapping of the presented process to a tuplespace-based application would result in six different tuplespaces, since the Petri net uses

[1] Similar to the ϵ tuple in EWFNs.

six distinct places. It is clear however that using so many different tuplespaces – each running as a separate operating system process with the need of being deployed and configured properly – bears considerable management and maintenance efforts; not to mention more complex Petri nets with a far greater number of places. Moreover, each of the tuplespaces would be used very inefficiently since it would only contain a very small number of tuples at a time. One of the main features of tuplespaces is the ability to provide means for multiple clients to coordinate their work based on sharing state in the form of tuples. Consequently, it should be possible to collapse all places of the original Petri net into one single tuplespace shared by each of the client applications. The main challenge for this approach is to encode the structure of the original Petri net, i.e. places, transitions and the arcs between them, using the tuplespace model.

We will investigate two possibilities for this approach, and (i) provide algorithms for the transformation of Petri nets to the tuple model, and (ii) show algorithms to implement the coordination logic of the tuplespace client applications that resemble individual transitions of the original Petri net. The transformations we present transform the graph itself, i.e. they operate on the level of the Petri net and produce and consume tuples from a shared tuplespace. The operational semantics is implemented by tuplespace client applications. The basis for our algorithms are Place-Transition Nets [Rei85], since each non-hierarchical CP-net can be transformed into a semantically equivalent PT-net [Jen92]. As a consequence, our approach allows to execute *any* Petri net dialect for which a transformation to a PT-net exists. For better readability, we repeat the definition of PT-nets here; it is however the same as Definition 28 in Chapter 4.

Definition 41 (Identical to Definition 28)**.** *A Place-Transition Net is a tuple* $PN = \langle P, T, F, W, I_0 \rangle$ *with:*

- P *a finite, non-empty set of* places
- T *a finite, non-empty set of* transitions *with* $P \cap T = \emptyset$
- $F \subseteq (P \times T) \cup (T \times P)$ *a set of arcs known as the* flow relation
- $W : F \rightarrow \mathbb{N}^*$ *the* weight-function *assigning a weight to each arc* $f \in F$,

denoting the number of tokens produced / consumed by a transition

- $I_0 : P \rightarrow \mathbb{N}_0$ *the* initialization function *that assigns the initial number of tokens to a place such that* $\forall p \in P : I_0(p) \in \mathbb{N}_0$

For better readability, the following algorithms use simplified versions of Linda operations from the JavaSpaces interface. Notifications (via `notify()`) were added to allow for optimizations in certain algorithms. In these cases, a "pure Linda" version of the algorithm is presented before introducing the optimization using notifications. Note that the operations are also supported by our own tuplespace implementation presented in Chapter 7.

Figure 6.3 presents an UML Class Diagram of the operations used in the algorithms. Class `Space` offers the tuplespace operations used in the Algorithms. The operation names are the same as in JavaSpaces, their parameters however have been simplified: operation `Space.Take(template)` for instance models the JavaSpace operation `take(entry, transaction, timeout)`. Tuple consuming operations (`take` and `read`) are always blocking, i.e. their `timeout` parameter is supplied with a positive number.

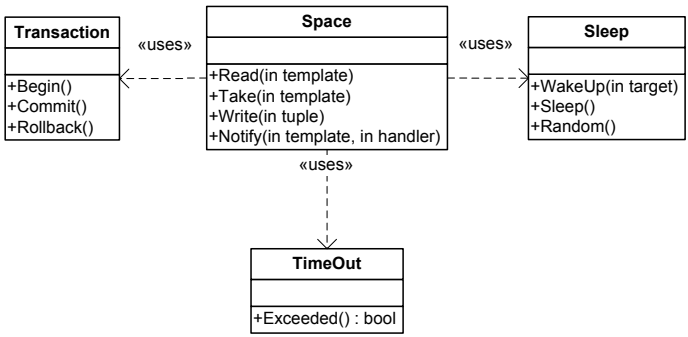


Figure 6.3: Simplified JavaSpaces Operations Used in the Algorithms

Transactions and timeouts are modeled as separate classes and a `Sleep` helper class was added. The methods in class `Transaction` follow the semantics of JavaSpaces transactions, i.e. they provide support for ACID transactions through the methods `begin()`, `commit()` and `rollback()`. Class `Timeout`

manages the timeout of tuplespace operations and transactions, the method `Exceeded()` returns `true` if the timeout has been reached. Class `Sleep` is a helper class that suspends the current thread of execution until being woken up (method `Sleep()`), or for a random amount of time (method `random()`). A currently sleeping thread can be woken up by calling `WakeUp()` on the associated `Sleep` object.

6.1.2 Model 1 - One Tuple per Token

The central idea behind this approach is to model each *token* in the original Petri net as a separate tuple; the place they reside in is encoded by the tuple itself. A tuplespace is thus initialized by creating tuples of the form

$$\langle \text{NetID, PlaceID} \rangle$$

for each place and each token a particular place contains (Algorithm 6.1). NetID is an identifier for the Petri net to be executed, allowing to distinguish between tuples from other Petri nets already stored in the tuplespace, e.g. to be able to execute more than one Petri net using a single tuplespace. When implementing this algorithm, an InstanceID should also be added to each tuple, allowing not only different Petri nets to be distinguished on a single tuplespace, but also different, concurrently running instances of the same Petri net model. For clarity of the discussions to follow however, we omit this simple enhancement.

Algorithm 6.1 Initialization: One Tuple per Token

```
for all p ∈ P do
    for i ← I_0(p) down to 1 do
        SPACE.WRITE(⟨NetID, p⟩)
    end for
end for
```

The logic executed by tuplespace clients to implement Petri net transition semantics for this encoding is presented in Algorithm 6.2. The tuplespace client uses a transaction (see position (2) in Algorithm 6.2) and blocking `take`

operations (see position (3)) to atomically consume a token for each incoming arc. The template employed to identify tuples on incoming places specifies that only those tuples match that reside in a place that is a source place of the transition, i.e. $\pi_1(x) = p, \langle p, t' \rangle \in (P \times T) \cap F$ with t' being the transition to be modeled. To cope with the situation that more than one tuplespace client waits for the same tuple to arrive (called "choice" or "conflict" in Petri net terms), a transaction with random timeout around the tuple consumption operations (see the lines marked with (2), (5) and (6) in Algorithm 6.2) ensures that no deadlock can occur. If the transaction times out, all tuples consumed and produced are given back to the tuplespace and the transition sleeps for a random amount of time. This allows other, conflicting transitions to consume the tuples and thus the deadlock is resolved. After the random timeout, the transition algorithm begins from the start. This of course is the most primitive approach to resolve deadlocks caused by consuming conflicting tuples, and was only used for simplicity of the algorithm. In practice, more mature algorithms for deadlock resolution, e.g. as proposed in [SKS98] should be used. After the tuples have been successfully consumed and thus the enablement rule of the transition has been satisfied, the transition executes its business logic and then writes tuples, according to its outgoing arcs and their weight, back to the tuplespace (see position (4)). When writing the tuples back to the space, their PlaceID changes to the ID of the place that is at the destination of the outgoing arc of the transition.

Algorithm 6.2 Transition: One Tuple per Token

Require: t' : me, i.e. the transition that is modeled by this algorithm.
Require: NetID:, the unique identifier of the Petri net.

```
loop                                                                    (1)
    TRANSACTION.BEGIN                                                   (2)
    for each x ∈ {⟨p, t⟩ ∈ (P × T) ∩ F | t = t'} do
        for i ← W(x) down to 1 do
            SPACE.TAKE(⟨NetID, π1(x)⟩)                                  (3)
        end for
    end for

    for each x ∈ {⟨t, p⟩ ∈ (T × P) ∩ F | t = t'} do
        for i ← W(x) down to 1 do
            SPACE.WRITE(⟨NetID, π2(x)⟩)                                 (4)
        end for
    end for

    if not TIMEOUT.EXCEEDED then
        TRANSACTION.COMMIT                                              (5)
    else
        TRANSACTION.ROLLBACK; SLEEP.RANDOM                              (6)
    end if
end loop
```

Note that an efficient solution specifically tailored to the problem of conflicts is discussed in Section 6.4, together with a deeper analysis of the problem of overlapping `sync` operations as we will then call this situation.

6.1.3 Model 2 - One Tuple per Place

In this approach, we represent each *place* of the original Petri net by a tuple instead of generating a tuple for each token. The *place tuple* therefore contains a field reflecting the amount of tokens currently held by the place. This is the reason why the algorithm (Algorithm 6.3) to initialize the tuplespace becomes

very simple: for each place, a tuple in the form

$$\langle \text{NetID, PlaceID, NumTokens} \rangle$$

is written to the space.

Algorithm 6.3 Initialization: One Tuple per Place

for each $p \in P$ **do**
  SPACE.WRITE($\langle$NetID, $p, I_0(p)\rangle$)
end for

The algorithm for implementing transition semantics for this encoding however becomes slightly more complex. Algorithm 6.4 uses four instead of two tuplespace operations per iteration compared to Algorithm 6.2. Each arc (incoming and outgoing) requires to consume a corresponding place tuple (see lines (2) and (3)), modifying it (adding or removing a number of tokens) and writing it back to the tuplespace. The algorithm also requires predicate support in templates to support the transition enablement rule: a place tuple may only be consumed if it contains equal or more tuples than the weight of the corresponding incoming arc of the transition (formalized by the expression "$\geq W(x)$" in the third position of the template tuple on line (2)). Moreover, line (3) in the algorithm requires support for a wildcard character[1] in the underlying tuplespace implementation.

[1] Implemented by $*$ in EWFNs and *null* in JavaSpaces.

Algorithm 6.4 Transition: One Tuple per Place

Require: t' : me, i.e. the transition that is modeled by this algorithm.
Require: NetID:, the unique identifier of the Petri net.

```
loop                                                                    (1)
    TRANSACTION.BEGIN
    for each x ∈ {⟨p, t⟩ ∈ (P × T) ∩ F | t = t'} do
        tuple ← SPACE.TAKE(⟨NetID, π_1(x), ≥ W(x)⟩)                     (2)
        SPACE.WRITE(⟨NetID, π_1(x), π_3(tuple) − W(x)⟩)
    end for

    for each x ∈ {⟨t, p⟩ ∈ (T × P) ∩ F | t = t'} do
        tuple ← SPACE.TAKE(⟨NetID, π_2(x), *⟩)                          (3)
        SPACE.WRITE(⟨NetID, π_2(x), W(x) + π_3(tuple)⟩)
    end for

    if not TIMEOUT.EXCEEDED then
        TRANSACTION.COMMIT
    else
        TRANSACTION.ROLLBACK; SLEEP.RANDOM
    end if
end loop
```

6.1.4 Using the JavaSpaces Notification Facility

Algorithm 6.5 presents an alternative solution to Algorithm 6.4 that uses the JavaSpaces notification feature. Notifications are a JavaSpaces specific extension allowing a client to be notified when a certain template can be satisfied[1]. This algorithm reduces the chance for transaction rollbacks in the case where multiple transitions compete for the same tuple. Instead of sequentially waiting for tuples to be available on incoming arcs (e.g. waiting for the left-most place to the right-most place, each at a time) as in Algorithm 6.4, the tuplespace middleware asynchronously calls the `Handler` callback procedure to notify the transition if a tuple is available that matches one of the registered templates. If all templates are satisfied, the main loop of the algorithm is woken up (position (2)). The advantage over Algorithm 6.4 is that the `Sleep` operation is only woken up if all required tuples are reported to be available by the tuplespace middleware.

[1] Note that for simplicity of the algorithm, we assume that the notification handler is called for every matching tuple that is already present in the tuplespace, not only when a new tuple is inserted that matches the registered notifiation.

Algorithm 6.5 Transition: One Tuple per Place, Using the JavaSpaces Notification Facility

Require: t': me, i.e. the transition that is modeled by this algorithm.
Require: $M, N \subseteq (P \times T) \cap F$, sets of (incoming) arcs of a transition: M will hold arcs that are "satisfied" already, N will hold all incoming arcs of transition t', i.e. $N = \{\langle p, t\rangle \in (P \times T) \cap F \mid t = t'\}$.
Require: NetID: the unique identifier of the Petri net.

```
procedure HANDLER(x ∈ (P × T) ∩ F)
    M ← M ∪ x
    if M ∩ N = N then
        SLEEP.WAKEUP                                        //See (2)
    end if
end procedure

loop                                                        (1)
    M, N ← ∅
    for each x ∈ {⟨p, t⟩ ∈ (P × T) ∩ F | t = t'} do
        N ← N ∪ x
        SPACE.NOTIFY(⟨NetID, π1(x), ≥ W(x)⟩, HANDLER(x))
    end for
    SLEEP.SLEEP                                             (2)

    TRANSACTION.BEGIN
    for each x ∈ {⟨p, t⟩ ∈ (P × T) ∩ F | t = t'} do
        tuple ← SPACE.TAKE(⟨NetID, π1(x), ≥ W(x)⟩)
        SPACE.WRITE(⟨NetID, π1(x), π3(tuple) − W(x)⟩)
    end for

    for each x ∈ {⟨t, p⟩ ∈ (T × P) ∩ F | t = t'} do
        tuple ← SPACE.TAKE(⟨NetID, π2(x), ∗⟩)
        SPACE.WRITE(⟨NetID, π2(x), W(x) + π3(tuple)⟩)
    end for

    if not TIMEOUT.EXCEEDED then
        TRANSACTION.COMMIT
    else
        TRANSACTION.ROLLBACK; SLEEP.RANDOM
    end if
end loop
```

6.1.5 Qualitative Evaluation of Model 1 and Model 2

Both models presented have their strengths and weaknesses. In order to get a better understanding of their properties, we asses them based on the criteria (i) number of tuples required, (ii) operations per transaction, (iii) likelihood of transaction rollback, (iv) predicates in templates required and (v) the ability to directly execute CP-nets.

Number of tuples required Each encoding model produces a different amount of tuples when representing a Petri net. In model 1, the number of tuples depends on the number of tokens present in the Petri net since each token is represented as a separate tuple. Model 2 in contrast represents places as tuples, the number of tuples thus is constant as tokens are represented as numbers in the place tuple. It highly depends of the type of Petri net (i.e. if it is *bounded*) which encoding generates more tuples.

Operations per transaction The amount of tuples required is a direct indicator for the number of operations executed within a single transaction of a transition. The more tuples need to be consumed, the more operations are required within the transaction and consequently the longer the transaction actually needs to run. For this category, model 2 is better than to model 1 as all required tuples from a particular place can be consumed at once within one operation, since tuples are represented by numbers in the place tuple. Removing three tuples from the same place requires to consume the place tuple, modify it and write it back to the tuplespace. In model 1, the same task requires three operations since tokens are tuples; each tuple needs to be consumed separately from the tuplespace. Note that the same argument applies to the task of a transition to generate tokens as well.

Likelihood of transaction rollback Transaction rollbacks are used to resolve conflicts, i.e. two or more transitions that want to consume the same tuple(s) from shared places (see Section 6.4 for a detailed discussion of this problem). Model 2 generally has a higher likelihood of transaction rollbacks since operations on the tuplespace are of coarser granularity. In

model 2, a place is consumed as a whole instead of a token at a time as in model 1. Consider e.g. the case where two transitions want to consume tokens from the same place (i.e. they are in conflict), only one transition wins; the other transition rolls its transaction back although both could be satisfied when enough tokens are available in the place. This situation cannot happen in model 1 since each token is represented by a separate tuple; enough tuples would be available for both transitions and none of them would need to rollback its transaction.

Required predicates in templates In contrast to model 1, the transition implementations of model 2 (shown by Algorithm 6.4) require predicate support in the underlying tuplespace implementation, a feature that is not supported by many tuplespace systems. Model 1 in contrast does not require predicates in templates and only relies on identity matching, which is implemented by virtually all tuplespace systems.

Directly execute CP-nets Although CP-nets can be transformed to semantically equivalent PT-nets (see Section 7.3.1 for an example), only model 1 can be used to execute CP-nets *directly*. Token types (i.e. colors) can be directly mapped to tuples by representing each token type as a separate tuple with fields according to the definition of the token color. An example for this approach is depicted by Figure 6.4 in the next section. Here, the process is not translated to a PT-net before execution; it rather directly uses token colors from the original CP-net as tuple types (e.g. tuple req corresponds to color request). Model 2 always requires a PT-net for execution. Tuples are represented by numbers only (see e.g. field 3, NumTokens, in the place tuple), the encoding therefore cannot represent the type definition from a CP-net color.

Table 6.1 summarizes our findings when comparing model 1 and model 2 and their corresponding algorithms as presented in previous sections.

It is clear that if a CP-net is used for the definition of a process, model 1 is the approach of choice since it offers the possibility to execute it directly without the need to transform it into a PT-net. If however a standard PT-net needs to

Table 6.1: Evaluation of the Algorithms for Model 1 and Model 2

Criteria	**Model 1**	**Model 2**
Number of tuples required	depends	depends
Operations per transaction	many	few
Likelihood of transaction rollback	low	high
Predicates in templates required	no	yes
Directly execute CP-nets	yes	no

be executed (either obtained by transformation or available as PT-net directly), model 2 is the better choice – given that the tuplespace middleware chosen supports predicates in templates – since transactions generally tend to contain fewer operations. The quantitative evaluation of the presented algorithms in Section 7.3 will provide more details on this matter.

A fundamental problem of the presented algorithms however is that efficient conflict resolution for overlapping joins cannot be done without global knowledge about all template operations currently active on a given tuplespace. This is a problem all client-side implementations of synchronized tuple consumption operations have, thus we argue that the only reasonable place for this operation is on the tuplespace middleware itself. As a consequence, we revisit this problem in Section 6.4, where we propose a generic solution in form of a new member of the tuplespace coordination interface.

6.2 Expressing EWFNs in the Tuplespace Model

Similar to CP-nets, tuples are used as the token data structure in EWFNs. The approach of choice to execute EWFNs on a tuplespace therefore is model 1 - one tuple per token. In this section, we discuss details on how model 1 is employed to execute EWFNs and give an overview on the architecture of the resulting system. As in the previous section, we use a simple process inspired by the "Loan Approval" workflow from the WS-BPEL 2.0 specification [A+07] to exemplify our approach.

6.2.1 Architectural Concepts

Figure 6.4 shows the EWFN version of the "Loan Approval" CP-net in Figure 6.2. Since EWFNs and CP-nets share many concepts, the resulting models are very similar. It is important to note however that arcs in EWFNs carry the set of *types* from the tuples communicated over the arc, not an expression that describes what tuple will be produced by the given arc, as it is in CP-nets. An EWFN transition models a program that might be of arbitrary complexity, it would not make sense to put its source-code as annotation on the arcs. Transitions in EWFNs only show their "interface" in terms of the types of tuples consumed and produced.

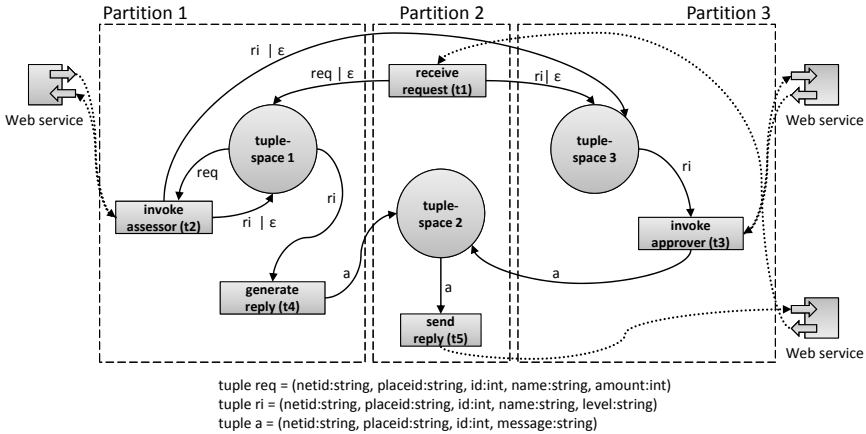


Figure 6.4: Loan Approval Flow Modeled as EWFN

Table 6.2: Tuple Definitions for the Loan Application EWFN (Figure 6.4)

Symbol	Definition
a	(id:int, message:string)
ri	(id:int, name:string, level:string)
req	(id:int, name:string, amount:int)

Figure 6.5 shows how the model described by Figure 6.4 which has been

transformed according to Algorithm 6.1 is deployed on a *single tuplespace*. Transitions are implemented by tuplespace client applications (boxes) that read and write tuples from and to a single, shared tuplespace represented by the circle in the center of the figure. Writing tuples (i.e. a `write` operation on the tuplespace) is depicted by an arc pointing towards the tuplespace, whereas reading (i.e. a `read` or `take` operation) a tuple is depicted as an arc leaving the tuplespace and pointing towards the client application. Arcs therefore model the direction of the flow of tuples. Each arc is annotated with a type, denoting the kind and format of the tuple actually communicated. Dotted lines denote Web service communication. The tuple structure from the original EWFN definition is modified by algorithm 6.1, i.e. the fields NetID and PlaceID were added to the original tuple definition from the EWFN.

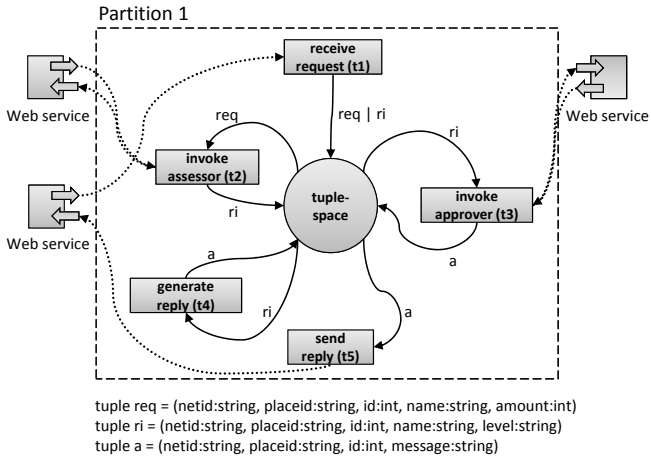


Figure 6.5: Loan Approval Application as Tuplespace-Based Application (Model 1)

It is important to note that tuplespace client applications not only implement the actual business logic (e.g. by invoking the internal credit rating service *invoke assessor*), but are also responsible for a part of the coordination logic (i.e. enabling directly succeeding activities). The value of the field *PlaceID:string*

of the tuple the *receive* client (t_1) writes into the tuplespace depends on the amount of the request, i.e. if it is greater than 10.000 the value is "invoke approver", else it is "invoke assessor". Subsequently, depending on the tuple written before, transition t_2 (in case t_1 wrote a req tuple) or t_3 (in case t_1 wrote a ri tuple) continues execution of the process.

The method of token passing for the coordination of individual activities the original process is composed of not only allows for centralized deployment as in Figure 6.5, but also for distributed deployment, as depicted in Figure 6.6. Instead of one central tuplespace, a separate tuplespace is used for each partition[1]. Note that this does not require any changes in the algorithms introduced so far; a separate parameter is added to each tuplespace operation that identifies the tuplespace to be used for reading or writing the tuple in question.

Different to the centralized deployment where all places of the original EWFN "collapsed" to a single place implemented by the central tuplespace, places in Figure 6.6 now mapped to three different tuplespaces. Control flow and data is passed between the partners involved by reading and writing tuples from and to one or more tuplespaces they are connected to. Previously local interactions such as passing control flow from the *generate reply* activity to *send reply* are now remote interactions, i.e. the *generate reply* activity writes a tuple into a remote tuplespace to pass control flow on to the next activity, which resides on a different partition than its predecessor.

Partitioning of the process is not limited to the example shown in Figure 6.6. In fact, it is only limited by the number of places in the original EWFN definition. The places of the EWFN definition are the vehicle for building individual partitions, they lay out the cutting lines the partitioning process may follow for building individual partitions. Note that the "wiring" of transitions and places is not changed by the partitioning process. It is only the places that are merged depending on the partitions created.

Naturally, creating a partition always means to introduce additional remote communication which is overhead that needs to be taken into account when

[1] In the following, the term "partition" identifies a certain part of the original process model that was split into multiple parts – the so called partitions.

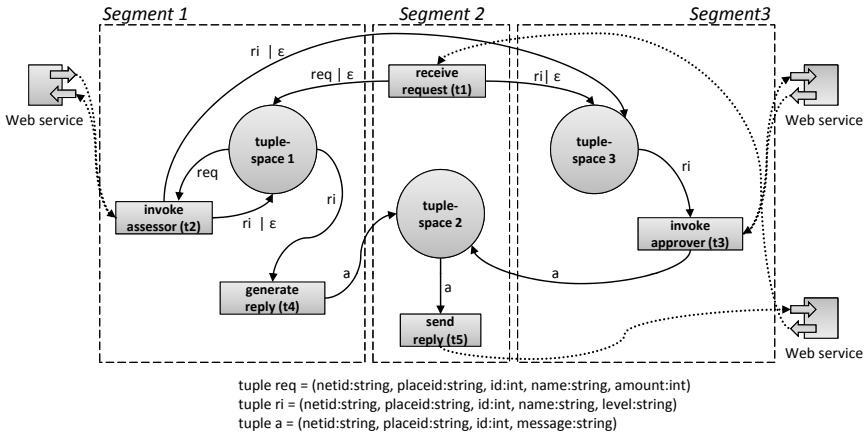


Figure 6.6: Loan Approval Application as Partitioned, Tuplespace-Based Application

deciding where to cut the original process and thus decide on the "shape" of each of the partitions. Moreover, partitioning may also split previously local transactions to distributed transactions, requiring additional efforts in order to ensure transactional semantics. The problem of finding optimal partitions of a process for a given IT infrastructure given the constraints discussed can be mapped to well known optimization problems. This is one of the major contributions of the work in [Wut10] and is therefore not further discussed here.

Clearly, data flow (and control flow to a lesser extend) of a process is the most influential property for the partitioning algorithm, as most network overhead is generated by passing data between partitions. Unfortunately, in many high-level workflow languages such as WS-BPEL, control flow can not only be expressed explicitly through links, but also implicitly by using structured activities. Moreover, data flow in BPEL is expressed entirely implicit through shared variables. This mix of explicit and implicit expression of data and control flow together with the functionality of Dead-Path-Elimination (DPE) [A+07]

makes BPEL particularly hard for decentralized execution. Complex analysis techniques are required to derive explicit data and control flow from a given BPEL process before algorithms for partitioning can be applied [KKL08, Kha07].

The advantage of EWFNs in the context of process partitioning is that both data- and control flow are explicit parts of the EWFN execution model. Algorithms for partitioning therefore can be applied *directly* to the EWFN without the need for preliminary data flow analysis [Wut10].

6.2.2 Discussion

As discussed, places of an EWFN essentially form the "cutting lines" of how a process may be split into different partitions. The atomic units distributed to different partitions therefore are the transitions of the EWFN. As this is however a very low level where partitioning might allow particular problematic configurations (such as distributing the individual transitions a BPEL `receive` operation is composed of), the partition granularity of our prototype workflow engine is increased to the level of composite transitions (cf. Section 5.1.1.3), i.e. the EWFN pattern of a BPEL activity is represented as a single, composite transition, implemented by a tuplespace client program. This type of transition represents the atomic unit for partitioning in the prototype. Please note however that this is a pure implementation issue, the concept of EWFNs allows the partition granularity to be at "EWFN atomic" level, i.e. the level where tuplespace client programs implement individual, not composite EWFN transitions.

As a result, the role of places in an EWFN from an execution point of view is rather to indicate buffered (possible remote-) communication and therefore mark borders of individual partitions, than to resemble a tuplespace. Using the algorithms presented in Section 6.1 the structure of the EWFN is encoded in the tuples communicated, making them *self-contained*, i.e. they contain the information in which place they currently reside in. Such a tuple has no relationship to the place it is stored, the partitioning process therefore can freely merge places without changing the semantics of the deployed EWFN. This is the key principle behind our method how to deploy EWFNs to an arbitrary

number of machines. Places can be freely merged so that all places that are assigned to the same machine are deployed using one single tuplespace.

Based on the aforementioned method of arbitrary overlapping places depending on the infrastructure to deploy to (e.g. the number of machines available, their available computing power and the capacity of their network connections), a whole spectrum of different deployments for a single process model is supported. Figure 6.7 highlights different deployment scenarios covering the full spectrum of deployment possibilities using our method. The deployment of the process model is visualized using an abstract notation: circles represent activities, solid arrows represent tuplespace communication and dashed arrows represent Web service communication. Communicating data- and control flow within the process is facilitated through communication with tuplespaces, which can be both local or remote, e.g. passing control flow between activities deployed on the same partner versus passing control flow between activities deployed on different partners. This is similar to Web service communication: the concept of a *binding* [WCL+05] allows to change the transport medium used to invoke a certain service independent of the service's functionality. This way Web services can be reached through different communication or transport mechanisms. In the course of our work, we developed different tuplespace Web service bindings [WML08a, WML08c, Sch07], allowing to use tuplespaces instead of e.g. HTTP as the transport mechanism to invoke a Web service. A solid arrow between an activity and a Web service in Figure 6.7 means that the Web service is invoked through the tuplespace binding. As a result, all interactions at a partner (between activities themselves and between activities and Web services) may be carried out in a homogeneous manner using tuplespaces as the transport mechanism.

Figure 6.7 highlights five distinct positions in the "decentralization spectrum" of possible deployments for a given process model:

Scenario 1: centralized orchestration The left hand side of the spectrum shows the process model being centrally enacted: the orchestration logic of the process is executed at a single partner (P1), local tuplespace communication is employed between individual activities, and Web service

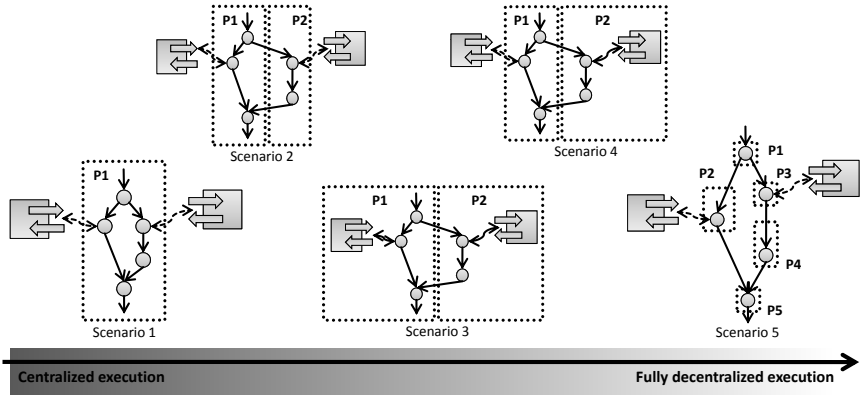


Figure 6.7: Range of Possible Deployments with our Approach

interaction between activities and their Web service implementations. The whole process model is executed by one single partner.

Scenario 2: partitioned orchestration In this scenario, execution of the orchestration logic of the process is carried out by two partners, P1 and P2. Local tuplespace communication is employed within the process partition of each partner and remote tuplespace communication happens between them. Communication between activities and their implementations is unchanged compared to scenario 1, i.e. Web service communication is used.

Scenario 3: partitioned orchestration, homogeneous WS communication Although the shape of the partitions containing the orchestration logic has not changed, the services providing activity implementations are now invoked using the same communication mechanism that is used within and between partitions. In this scenario, the only communication mechanism used is provided by tuplespaces, i.e. communication between process activities themselves as well as communication between activities and Web services is carried out using tuplespaces.

Scenario 4: partitioned orchestration, heterogeneous WS communication The deployment scenario in this position combines previous scenarios 2 and 3. Here, local tuplespace-based interaction using the tuplespace Web service binding is carried out with one service at P2. At P1 however, the Web service is invoked using a non-tuplespace-based transport.

Scenario 5: maximally decentralized process This scenario forms the far end of the distribution spectrum, showcasing the maximal possible level of decentralization using our approach. Each activity of the orchestration logic and the Web service implementations reside at different partners. As a result, only remote communication is carried out, using tuplespace interaction for coordinating individual activities and Web service or tuplespace-based communication to invoke Web service implementations.

6.3 Algorithms on Tuplespaces for EWFN Execution

After discussing the mapping of EWFNs to tuples and how the conceptual architecture of a workflow system based on these ideas can be realized, this section describes algorithms for the implementation of a tuplespace system that "natively" executes EWFNs. This is facilitated by providing a one-to-one mapping of each different arc in the EWFN definition presented in Chapter 4 and operations in the proposed tuplespace implementation. As a result, all aspects of the theoretic EWFN model are covered by a tuplespace implementation. We start by discussing the initial (Linda) set of operations, namely `read` (Section 6.3.3), `write` (Section 6.3.2) and `take` (Section 6.3.5), and then present algorithms that implement the extension arcs, i.e. `readall` (Section 6.3.4), `takeall` (Section 6.3.6) and `update` (Section 6.3.7). The `sync` operation introduced in Chapter 5 will be covered in a separate section since its implementation is especially challenging. Altogether, the algorithms presented build the basis for the tuplespace kernel prototype that is discussed and evaluated in Chapter 7.

6.3.1 Symbols and Data Structures Used

Table 6.3 introduces additional symbols and notations used in the description of the algorithms, which is based on the notation introduced in Chapter 4. An overview of all symbols and functions used in this work is given in the appendix.

Table 6.3: Notations used

Notation	Meaning
$\pi_i(t)$	Returns a projection to the ith element of a tuple t
$\mathcal{P}(S)$	Denotes the power set of a set S
F^{MS}	F^{MS} is a multiset of tuples denoting the contents of a tuplespace
$tu \in \Sigma$	A tuple
$te \in X$	A template
$OP = \{\text{“}read\text{”}, \text{“}take\text{”}\}$	The set of possible operations to apply a template to the tuplespace
$K \subseteq X \times OP$	The set of parameter tuples given to a `sync` operation. $\langle te, op \rangle \in K$ is one parameter with op denoting the operation to use to apply template te
$H \subseteq X \times OP \times \mathbb{B}$	The set of active templates on the tuplespace, described in tuple form $\langle te, op, b \rangle$, $b \in \mathbb{B} = \{\texttt{true}, \texttt{false}\}$ indicates if the template $te \in X$ can be satisfied
$G \subset \mathcal{P}(H)$	The set of sets of active templates, more specifically, the parameter sets of all tuple consuming operations currently active on the space. A `sync` operation with three templates for instance results in a set with three elements that is added to set G
$U \subseteq H$	The temporary set used to construct an element of set G

$\texttt{check}_\texttt{t} : X \rightarrow \mathbb{B}$	Checks if template $te \in X$ can be satisfied with the current state of the space by using the internal read function `Int.Read` $$\texttt{check}_\texttt{t}(te) = \begin{cases} \texttt{true}, & \text{if } \texttt{Int.Read}(te) \neq \epsilon \\ \texttt{false}, & \text{otherwise} \end{cases}$$
$\texttt{eval} : H \rightarrow \mathbb{N}$	An evaluation function that calculates a value used to select the blocking operation to wake-up if more than one may be woken up
$\texttt{max} : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$	Returns the maximum of two integers, i.e. $$\texttt{max}(n, m) = \begin{cases} n, & \text{if } n \geq m \\ m, & \text{otherwise} \end{cases}$$
$\texttt{count} : \Sigma \rightarrow \mathbb{N}$	Returns the highest number of equal elements in tuple $tu \in \Sigma$. Consider the tuple $tu = \langle 1, 1, 5, "a", "b" \rangle$ for instance, in this case $\texttt{count}(tu) = 2$
$L \subseteq X$	A set of templates with equal length (equal number of elements, i.e. $\exists n : \forall \text{ te} \in L : \lvert\text{te}\rvert = n$) for synchronized consumption. Note that the templates may contain join-variables
$\texttt{check}_{\texttt{t,jv}} : L \rightarrow \mathbb{B}$	The function that handles the matching for join-variables (cf. Algorithm 6.17)
$\circ$	The concatenation operator for tuples, for instance $\langle a, b \rangle \circ \langle 1, 2 \rangle = \langle a, b, 1, 2 \rangle$

6.3.2 Write

The `write` operation adds one element to the multiset that represents the tuplespace. In the original Linda model, `write` was called `out`. In terms of EWFNs, the `write` operation implements the `write` arc, i.e. an arc going from a transition to a place.

Algorithm 6.6 Write

Require: F^{MS}, multiset of tuples, i.e. the contents of the tuplespace.

function WRITE($tu \in \Sigma$)
  $F^{MS} \leftarrow F^{MS} + \{tu\}$
  SELECTANDWAKEUP(tu, "insert")
end function

Algorithm 6.6 presents the implementation of the `write` operation, consisting of the statement that adds the tuple to the multiset of existing tuples. By calling `SelectAndWakeUp`, the algorithm that detects conflicting tuple consumptions and implements callback mechanisms for blocked operations is notified that a new tuple is available (see Section 6.3.8 for details).

6.3.3 Read

The `read` operation (equivalent to Linda's `rd`) is structured into two algorithms: Algorithm 6.7 presents the external visible operation that is part of the coordination interface of our tuplespace implementation, whereas Algorithm 6.8 presents the internal read functionality, i.e. a module that is reused by other algorithms. As with all other operations, `read` has a corresponding `read` arc defined in the EWFN model.

Externally, the `read` operation comes in two fashions: blocking and non-blocking. If the user invoked the non-blocking version, the tuplespace is searched for a matching tuple using the internal `read` operation (Algorithm 6.8) and either a matching tuple or the empty tuple is returned. If the user invokes the blocking version however (i.e. the operation should block until a matching tuple is present in the space), Algorithm 6.7 adds the operation to the list of ac-

Algorithm 6.7 Read

Require: F^{MS}, multiset of tuples, i.e. the contents of the tuplespace.
Require: $G \subset \mathcal{P}(H)$, set of parameter sets of active operations on the tuplespace (see Algorithm 6.14).

```
function READ(te ∈ X)
    if non-blocking then return INT.READ(te)
    else
        G ← G ∪ {⟨te, "read", check_t(te)⟩}
        //parallel execution
        SLEEP.SLEEP || SELECTANDWAKEUP(null, null)
        R ← INT.READ(te)
        SPACE.UNLOCK
        return R
    end if
end function
```

tive tuple consumption operations and then waits until being woken up when a matching tuple is available. This is done using the function `SelectAndWakeUp`: after adding the template to the set of blocking operations via set G, it calls `SelectAndWakeUp` on a new thread and puts itself to sleep. The function `SelectAndWakeUp` (see Section 6.3.8 for details) evaluates all blocking operations and chooses one of the list of operations currently being satisfied to be woken up. Before waking up a specific operation, `SelectAndWakeUp` locks the tuplespace to avoid race conditions. After being woken up, the read operation can read the tuple and unlock the tuplespace.

Algorithm 6.8 presents the internal version of the `read` operation. Here, the multiset of tuples representing the tuplespace is searched until the first matching tuple for the given template is found. The algorithm then immediately returns this tuple and exits. Tuple matching is done using the $\approx$ relation as defined in Section 4.2.1.

Algorithm 6.8 Int.Read

Require: F^{MS}, multiset of tuples, i.e. the contents of the tuplespace.

```
function INT.READ(te ∈ X)
    for all tu ∈ F^MS do
        if ⟨tu, te⟩ ∈ ≈ then
            return tu
        end if
    end for
    return ϵ
end function
```

6.3.4 Readall

In contrast to the `read` operation, `readall` returns all matching tuples that are currently in the tuplespace. `readall` is always non-blocking, i.e. if no matching tuple is available, the operation returns the empty tuple. The `readall` operation is the counterpart of the `readall` EWFN arc presented in Section 4.5.1

Algorithm 6.9 Readall

Require: F^{MS}, multiset of tuples, i.e. the contents of the tuplespace.
Require: R^{MS}, multiset of matching tuples to be returned.

```
function READALL(te ∈ X)
    R^MS ← ∅
    for all tu ∈ F^MS do
        if ⟨tu, te⟩ ∈ ≈ then
            R^MS ← R^MS + {tu}
        end if
    end for
    if |R^MS| > 0 then return R^MS
    else return {ϵ} end if
end function
```

Algorithm 6.9 presents the implementation of the `readall` operation. It iterates through all tuples in the tuplespace and adds matching tuples to the

multiset R^{MS} which is returned when the entire tuplespace has been searched.

6.3.5 Take

In a nutshell, the `take` operation is the *destructive* version of `read`, i.e. instead of returning a copy, the matching tuple is removed from the tuplespace and given back to the client. Similar to the `read` operation, `take` is also divided into two algorithms, one presenting the logic for the externally visible `take` operation implementing the logic for blocking or non-blocking operation, the other one presenting reusable, internal logic that implements the actual operation on the tuplespace level. The `take` operation in EWFNs is represented by the default type of arcs, i.e. a solid line with one arrow-head.

Algorithm 6.10 Take

Require: $G \subset \mathcal{P}(H)$, set of parameter sets of active operations on the tuplespace (see Algorithm 6.14).

```
function TAKE(te ∈ X)
    if non-blocking then return INT.TAKE(te)
    else
        G ← G ∪ {⟨te, "take", check_t(te)⟩}
        //parallel execution
        SLEEP.SLEEP || SELECTANDWAKEUP(null, null)
        R ← INT.TAKE(te)
        SPACE.UNLOCK
        return R
    end if
end function
```

In non-blocking mode, Algorithm 6.10 calls function `Int.Take` and directly returns the result. In blocking mode however, the operation is scheduled for later wake-up by adding its parameters to set G, calling function `SelectAndWakeUp` and suspending operation until woken up.

Note that `SelectAndWakeUp` has two modes of operation: (i) when being called with "null" arguments, it is in "wakeup mode", i.e. its only task is to check if there are operations eligible to be woken up. (ii) when being called

with arguments, it is in "state update mode", i.e. the state of the tuplespace has changed (either a tuple has been added or removed) and thus new operations might be eligible to be woken up.

Algorithm 6.11 Int.Take

Require: F^{MS}, multiset of tuples, i.e. the contents of the tuplespace.

```
function INT.TAKE(te ∈ X)
    for all tu ∈ F^MS do
        if ⟨tu, te⟩ ∈ ≈ then
            F^MS ← F^MS − {tu}
            SELECTANDWAKEUP(tu, "remove")
            return tu
        end if
    end for
    return ϵ
end function
```

Algorithm 6.11 presents the internal version of the `take` operation. It iterates over the tuples in the tuplespace and returns the first match that was found.

6.3.6 Takeall

The `takeall` operation is the destructive version of `readall`, i.e. it is non-blocking and removes, instead of copies, all matching tuples from the space. It is the counterpart of the `takeall` arc presented in Section 4.5.2. Note that `takeall` can also be used as a multi-delete by ignoring the values returned by the operation. This way, tuple removal as required by the EWFN implementation of BPEL's `exit` activity for instance can be implemented.

Algorithm 6.12 presents the implementation of the `takeall` operation. It iterates over all tuples in the tuplespace, removes matching tuples and adds them to the multiset R^{MS} which stores the result.

Algorithm 6.12 Takeall

Require: F^{MS}, multiset of tuples, i.e. the contents of the tuplespace.
Require: R^{MS}, multiset of matching tuples.

```
function TakeAll(te ∈ X)
    R^MS ← ∅
    for all tu ∈ F^MS do
        if ⟨tu, te⟩ ∈ ≈ then
            R^MS ← R^MS + {tu}
            F^MS ← F^MS − {tu}
            SelectAndWakeUp(tu, "remove")
        end if
    end for
    if |R^MS| > 0 then return R^MS
    else return {ϵ} end if
end function
```

6.3.7 Update

The `update` operation essentially is a sequence of `take` and `write`, that avoids unnecessary shipping of the result of the `take` operation back to the client. The operation is non-blocking, i.e. if no matching tuple is found, control is immediately given back to the client. The `update` operation implements the EWFN `update` arcs as defined in Section 4.5.3.

Algorithm 6.13 shows how the `update` operation is implemented. It essentially consists of a `take` followed by a `write`, removing the tuple matching the given template and adding the new tuple to the space. If no matching tuple is found, the operation has no effect and returns immediately with the empty tuple.

6.3.8 SelectAndWakeUp

The `SelectAndWakeUp` algorithm is used whenever an operation needs to block and then woken up when a matching tuple (or a set of matching tuples) is available. For that, it implements a callback mechanism that queues all

Algorithm 6.13 Update

Require: F^{MS}, multiset of tuples, i.e. the contents of the tuplespace.

```
function UPDATE(te ∈ X, tu' ∈ Σ)
    for all tu ∈ F^MS do
        if ⟨tu, te⟩ ∈ ≈ then
            F^MS ← F^MS − {tu}
            F^MS ← F^MS + {tu'}
            SELECTANDWAKEUP(tu, "remove")
            SELECTANDWAKEUP(tu', "insert")
            return tu
        end if
    end for
    return ϵ
end function
```

blocking operations and selects one (or *all* in case of `read`) of a list of wake-up candidates to continue operation. This algorithm also plays an important role when it comes to resolve conflicts of overlapping `sync` or `take` operations (see Section 6.4.1 for details). Note that `SelectAndWakeUp` is an internally used algorithm, i.e. it is not part of the coordination interface of the tuplespace.

The central idea of the algorithm is to maintain a set $G \subset \mathcal{P}(H)$ on a per space basis, that contains the parameter sets $p^\star \in G$ of all blocking, tuple consuming operations. The maintenance of set G is implemented by the first part (starting from position (1)) of Algorithm 6.14. Each of the tuples $tu = (te, op, b) \in H$ contain a boolean $b \in \mathbb{B}$ indicating if template $te \in X$ can be satisfied under the current state of the space. The boolean values are updated by intercepting tuple insertion and removal operations and calling `SelectAndWakeUp` with the tuple that was added or removed in order to be able to modify set G accordingly. Set G contains global knowledge about all blocking tuple consuming operations on the space and can therefore be used to resolve deadlocks caused by overlapping `sync` operations as described in Section 6.4.1. Note that `SelectAndWakeUp` handles both, single blocking operations (`read` or `take`, with $\left|p^\star\right| = 1$) as well as `sync` operations with

multiple templates ($\left|p^\star\right| > 1$) and join-variables involved (see Section 6.4). Operations electable for wake-up (operations whose templates can *all* be fulfilled with the current state of the space i.e. $\forall k \in p^\star : \pi_3(k) = \texttt{true}$, see position (2)) are evaluated using the function `eval` at position (3). Note that Algorithm 6.14 does not show how identical tuples residing in the tuplespace are handled: generally, it is necessary to store the number of matching tuples together with a blocking template in order to be able to wake-up as many blocking operations as there are matching tuples available. Further note that in practice, the value `best` is not only an integer value denoting the score of a certain operation but also a reference to the operation itself.

The implementation of the `eval` function is intentionally left open since it influences the behavior of the tuplespace significantly. In the following, we provide examples for possible implementations, some of them require additional metadata to be stored alongside each parameter set $p^\star \in G$:

Time-based selection Select the operation that is member of the set G for the longest time: additional metadata stored in set G is a time-stamp indicating when the operation was added. `eval` in this case returns the delta between the current time and the time of addition of the operation to set G.

Custom priority Select the operation that has the highest priority: on EWFN deployment time for instance, this allows for custom definition of operations that are required to have precedence over others in case of conflicting tuple consumption. `eval` in this case returns the priority value assigned to the respective operation – higher values lead to earlier selection for wake-up.

Throughput The operation with the least number of templates wins: in order minimize the average time an operation waits, the operation of a set of conflicting operations is woken up that has the least number of templates and thus consumes the least number of tuples. As a result, more blocked operations with fewer templates might be electable to be woken up instead of just one operation with many templates. `eval` in this case

Algorithm 6.14 SelectAndWakeUp

Require: $G \subset \mathcal{P}(H)$, The parameter sets of sleeping, tuple consuming operations.
Require: $V \subseteq X$, temporary set to construct the input for $\texttt{check}_{\texttt{t,jv}}$.

```
procedure SelectAndWakeUp(tu ∈ Σ, op ∈ {"insert", "remove"})
    Space.Lock, best ← 0
    for all p* ∈ G do
        if tu ≠ null then                                          (1)
            if |p*| > 1 ∧ ∃k ∈ p* ∃x ∈ π1(k) : x is a join-variable then
                ∀k ∈ p* : V ← V ∪ π1(k)
                if check_t,jv(V) = true then
                    ∀k ∈ p* : π3(k) ← true
                else
                    ∀k ∈ p* : π3(k) ← false
                end if
            else
                for all k ∈ p* do
                    if ⟨tu, π1(k)⟩ ∈ ≈ ∧ op = "insert" then
                        π3(k) ← true
                    else if ⟨tu, π1(k)⟩ ∈ ≈ ∧ op = "remove" then
                        π3(k) ← false
                    end if
                end for
            end if
        end if
        if ∀k ∈ p* : π3(k) = true then                             (2)
            best ← max(best, eval(p*))                             (3)
        end if
    end for
    if best > 0 then WakeUp best                                   (4)
    else Space.Unlock end if
end procedure
```

returns the result of the expression $\left|p^\star\right|$.

Random Randomly choose one among the list of conflicting operations to

wake-up: this strategy may be used as a default if no strategy is given in the EWFN definition. `eval` in this case returns a random number.

Generally, the `eval` function should put a preference on `read` over `take` operations, i.e. all operations that have no impact on the contents of the tuplespace should be woken up before operations that actually remove tuples.

6.4 Synchronized Tuple Consumption

Tuplespace implementations today do not offer operations that allow the synchronized consumption of multiple tuples at once, e.g. to implement a control flow join by waiting for a number of tuples (each representing one thread of control flow) to arrive. With state-of-the-art implementations, the logic that implements such a synchronizing join (a.k.a. AND-Join [RtHvdAM06]) has to be implemented by a client listening on the tuplespace that checks if *all* required tuples are available. It has to be ensured however that concurrent, "overlapping" join operations (where one join operation holds a tuple that another join operation would need and vice-versa) do not result in a deadlock. In this section, we show a solution for this problem in form of a new member of the coordination interface of our tuplespace implementation – the `sync` operation.

In case the tuples to be joined are distributed over a number of different remote tuplespaces, an efficient implementation of synchronized tuple consumption is more difficult to achieve. In this scenario, there is no single tuplespace which has global knowledge about potentially overlapping, concurrent join operations. Consequently, additional communication between the tuplespaces involved in the join operations or the tuplespace clients performing the join operations would be required. We solve this problem in form of a pattern presenting how tuples distributed over multiple tuplespaces can be consumed synchronously without requiring additional communication between the tuplespaces involved.

6.4.1 Motivation

Joining different paths of control flow is one particular use case that requires synchronized consumption of multiple tuples. More generally however, supporting synchronized tuple consumption is a prerequisite to execute Petri nets; recall that a transition is only enabled if all its input conditions (i.e. the source places of all incoming arcs contain at least one matching token) are fulfilled. Implementing a transition in form of a tuplespace client application thus requires to implement logic that blocks until all input conditions are fulfilled. In Section 6.1, this functionality was implemented in form of client-side logic that uses transactions and blocking tuplespace operations to reserve tuples and resolve conflicts by rolling these transactions back. In this section, we revisit this implementation and extract it as a separate, client-side operation. We then focus on developing an efficient solution that does not require transactions in form of a new tuplespace coordination primitive. A special case of synchronized tuple consumption is the implementation of a synchronizing join of control flow, which we will use as a run-through example throughout this section.

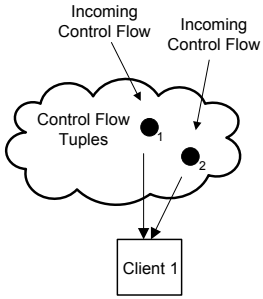


Figure 6.8: Simple Join

An implementation of a synchronizing join is depicted in Figure 6.8, where two threads of control, represented by one tuple for each concurrent execution path in one process instance[1], are joined into a single path of execution.

[1]Multiple control flow tuples belonging to the same process instance are correlated by a process instance identifier field in each control flow tuple.

The join node is implemented as tuplespace client (Client 1) and waits for the arrival of control flow tuples. Once these are available, the client can execute its business logic (i.e. the transition *fires* in Petri net jargon). Client 1 implements transition t' in Algorithm 6.2. The Algorithm operates directly on the Petri net definition, which also defines the number of incoming arcs that must be satisfied in order to be allowed to fire.

Algorithm 6.15 presents an implementation of the generic tuple synchronization logic using a state-of-the-art tuplespace system – SUNs JavaSpaces[1] in this case. The parameters of the algorithm are a set of tuples of the form $\langle te, op \rangle$, denoting if template $te \in X$ is used in either a non-destructive `read` or a destructive `take` operation. The general idea of this algorithm is to use the JavaSpaces notification feature to get notified if a tuple is inserted into the space that satisfies a template that is part of the synchronized tuple access operation (or `sync` operation for short). If all templates can be satisfied, the main algorithm is woken up (see (1)) and tries to execute all templates in order to form the resulting multiset R^{MS}.

The main task of this algorithm is to solve the problem of "overlapping templates" which is outlined in Figure 6.9: it depicts the case when `sync` operations of multiple clients share equal templates and thus compete for the same (set) of tuple(s) within a single operation. In the following, we list the three possible cases of overlapping templates in `sync` operations and discuss how Algorithm 6.15 handles each case.

One-Overlapping `sync` One-Overlapping `sync` operations are depicted in Figure 6.9(a). This situation occurs when two clients issue `sync` operations as implemented by Algorithm 6.15 that share one equal template. Consider the situation depicted in Figure 6.9(a), but with an empty tuplespace. When tuple 1 arrives, client 1 gets notified and registers its availability. Upon arrival of tuple 2, both clients get notified that a matching tuple has arrived, only client 1 however has matching tuples for all templates of its `sync` operation. Client 2 thus is still blocked while client 1 can proceed, consume both tuples and return. Consider now the

[1] `http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/`

Algorithm 6.15 `sync` Operation: Implementation using JavaSpaces

Require: $M, N \subseteq X$, Sets of templates: M holds templates that are satisfiable, N holds all templates of the operation.
Require: R^{MS}, Multiset of resulting tuples of this operation. Note that $\left|R^{MS}\right| = |K|$ since each template execution returns exactly one tuple.

```
procedure NotificationHandler(te ∈ X)
    M ← M ∪ te
    if M ∩ N = N then
        Sleep.WakeUp                                                    (1)
    end if
end procedure

function Sync(K ⊆ (X × OP))
    loop
        M, N ← ∅
        for all k ∈ K do
            N ← N ∪ π1(k)
            Space.Notify(π1(k), NotificationHandler(π1(k)))
        end for
        Sleep.Sleep                                                     (2)
        Transaction.Begin                                               (3)
        ∀k ∈ K : R^MS ← { R^MS + {Space.Take(π1(k))}, if π2(k) = "take"
                        { R^MS + {Space.Read(π1(k))}, otherwise
        if not TimeOut.Exceeded then
            Transaction.Commit                                          (4)
            return R^MS
        else
            Transaction.Rollback; Sleep.Random                          (5)
        end if
    end loop
end function
```

case where tuples 1, 2 and 3 arrive at the same time. Now, the main parts (starting from position (2)) of both `sync` operations are woken up at the same point in time and the client that consumes tuple number

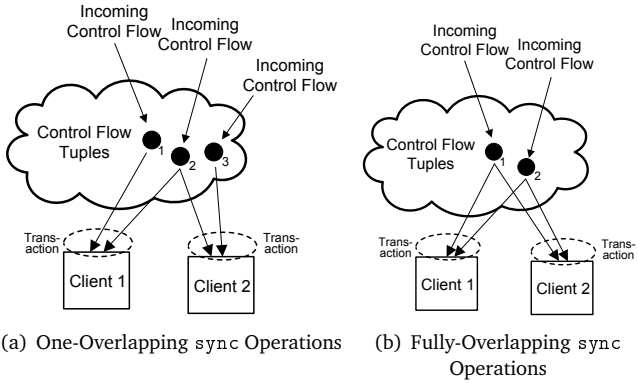


Figure 6.9: Overlapping `sync` Operations

two is the one that can proceed. The other client cannot fulfill the `sync` operation, thus the transaction times out, is rolled back (5) and all tuples consumed within the transaction are given back to the space.

Multi-Overlapping `sync` Multi-Overlapping `sync` operations denote a situation where the number of overlapping templates is greater than one but not equal to the total number of templates of a particular `sync` operation. In case of simultaneous arrival of all matching tuples, Algorithm 6.15 wakes up both `sync` operations from sleep (position (2)). It is now possible that during tuple consumption (position (3)), each client receives a part of the set of overlapping tuples. This creates a situation where it is impossible for both `sync` operations to fulfill all of their templates. A deadlock is the result which is resolved using random timeouts and retry as described shortly.

Fully-Overlapping `sync` Fully-Overlapping `sync` operations are depicted in Figure 6.9(b). They are a special case of multi-overlapping `sync` operations where *all* templates overlap. When matching tuples arrive causing both `sync` operations to be woken up, a deadlock is created. A deadlock is even the result when matching tuples arrive consecutively, since both

operations wake up at exactly the same time, i.e. when the last matching tuple arrives. This stays in contrast to the multi-overlapping case, where a deadlock is only created when both operations are woken up at the same time, which cannot happen there when tuples arrive consecutively, since there is at least one template that is not part of both operations.

Algorithm 6.15 resolves deadlocks by employing random timeouts to rollback a transaction if it is not committed in a given time interval. As a result, consumed tuples are given back to the tuplespace and made available for other conflicting transactions to be able to complete. As a result, the `sync` operation with the greatest timeout assigned will eventually be able to consume all requested tuples while the other operations will be blocked until a new set of matching tuples arrives.

Implementing the `sync` operation using coordination primitives of existing tuplespace middleware however has three inherent problems:

1. The algorithms presented use transactions as a means to reserve tuples and to cancel these reservations if there are conflicts. If atomic execution of a set of operations using a transaction does not return tuples for all templates, e.g. because a tuple has already been consumed by a conflicting `sync` operation, expensive conflict resolution strategies have to be implemented.
2. Network overhead is another problem of the algorithm presented: there are separate remote calls for template registration, notification and for the actual tuple consumption operations (`read` or `take`). In total, there are at least $3 * |K|$ network operations necessary for a single `sync` operation. If there are overlapping templates that are causing a conflict, its resolution involves additional network communication.
3. The core principle behind coordination languages, i.e. separating coordination logic from computation logic [GC92], is violated by Algorithm 6.15. The `sync` operation essentially is a low-level coordination primitive which was replicated by client-side logic, where only the computational logic should reside.

The fundamental problem all algorithms that use client-side logic to implement synchronized consumption of tuples suffer from, is that efficient conflict resolution cannot be done without global knowledge about all active template operations. We thus argue that the only reasonable place for the implementation of such operations is as part of the tuplespace middleware itself. Consequently, in the next section we introduce an algorithm for a server-side version of the `sync` operation, making it a "native" part of the coordination interface of a tuplespace middleware. Being located at the side of the tuplespace, the algorithm has access to all active tuple consuming operations and their templates, it thus can leverage global knowledge about all active templates to efficiently resolve conflicts. As a result, it does not suffer from the problems all client-side implementations of `sync` (such as e.g. Algorithm 6.15) inherently have. Especially, it does not rely on transactions for tuple reservations and transaction rollbacks for canceling these reservations.

6.4.2 The `sync` Coordination Primitive

Algorithm 6.16 presents the server-side implementation of the `sync` operation. The set $K \subseteq X \times OP$ represents parameters of the `sync` operation containing tuples of the form $\langle te, op \rangle$. Operation $op \in OP$ is the operation to be used with template $te \in X$. The first task of the algorithm is to check if each template te of tuple $\langle te, op \rangle \in K$ can be satisfied with the current state of the tuplespace. For this task, we use function $\texttt{check}_{\texttt{t,jv}}$, which is defined in Section 6.4.3. The resulting Boolean value is appended (via the concat operator $\circ$) to the tuple which is added to the set U. Set U is the annotated version of set K adding a boolean value indicating if a template can be satisfied of not. After updating set G, the algorithm goes to sleep after spawning a new thread that invokes the `SelectAndWakeUp` algorithm we discussed earlier, to ensure proper handling of the new additions to set G. If a `sync` operation was added that is satisfiable, `SelectAndWakeUp` decides if it is chosen to be woken up or not.

Note that, after being woken up (i.e. being notified that all requested tuples are available in the tuplespace), `sync` applies the set of templates (see position (2)) using the internal versions of `read` and `take`, (called `Int.Take`

Algorithm 6.16 `sync` Operation: Implementation as Coordination Primitive

Require: R^{MS}, the multiset of tuples returned by the `sync` operation, containing a tuple for each template in the parameter list. Note that $\left|R^{MS}\right| = \left|p^{\star}\right|$ since each template returns one tuple.
Require: $U \subseteq H$, temporary set used to construct an element of set G.
Require: $V \subseteq X$, temporary set to construct the input for $\texttt{check}_{\texttt{t,jv}}$.
Require: $G \subset \mathcal{P}(H)$, set of parameter sets of active operations on the tuplespace (see Algorithm 6.14).

function SYNC($K \subseteq X \times OP$)
  $U, V, R^{MS} \leftarrow \emptyset$
  $\forall k \in K : V \leftarrow V \cup \pi_1(k)$
  if $\texttt{check}_{\texttt{t,jv}}(V) = \texttt{true}$ **then**
    $\forall k \in K : U \leftarrow U \cup (k \circ \texttt{true})$
  else
    $\forall k \in K : U \leftarrow U \cup (k \circ \texttt{false})$
  end if
  $G \leftarrow G \cup U$ (1)
  //*parallel* execution
  SLEEP.SLEEP || SELECTANDWAKEUP($null, null$)
  $$\forall k \in K : R^{MS} \leftarrow \begin{cases} R^{MS} + \{\text{INT.TAKE}(\pi_1(k))\}, \text{if } \pi_2(k) = \text{“take”} \\ R^{MS} + \{\text{INT.READ}(\pi_1(k))\}, \text{otherwise} \end{cases} \quad (2)$$
  SPACE.UNLOCK
  return R^{MS}
end function

and `Int.Read` in Section 6.3). Using a lock on the tuplespace ensures that all templates can be satisfied once the operation is woken up (i.e. no races occur).

Algorithm 6.16 is deadlock free as shown by Figure 6.10: after it puts itself to sleep and at the same time called `SelectAndWakeUp` (abbreviated as “SaW” in the figure), both paths after the `Lock` operation lead to an `Unlock`. The dashed line on the left-hand side of the figure indicates that the `Sleep` operation is woken up by the `WakeUp` operation, thus leading to an `Unlock` operation after the branch `best>0` was taken. An evaluation of the `sync` operation implemented by Algorithm 6.16 is presented together with the architecture of

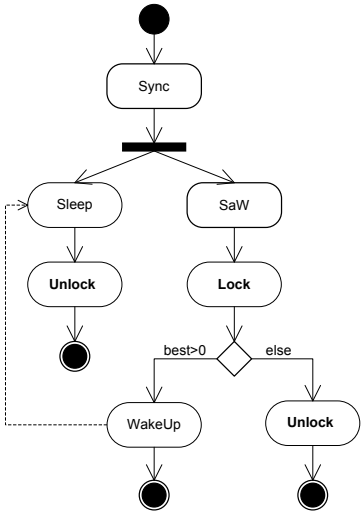


Figure 6.10: Tuplespace Locking by the `sync` Primitive

our proposed tuplespace kernel in Chapter 7.

There are cases where a given Petri net model does not require conflict resolution in `sync` operations: if there are no overlapping joins in the Petri net, no conflicts can occur and thus the complex mechanism to maintain set G is not required.

In case of a PT-net, the existence of overlapping joins can be detected by statical analysis, i.e. by checking if the sets of input places of all transitions are pair-wise disjoint.

Definition 42 (conflict-free PT-net)**.** *A PT-net is called* conflict-free *iff*

$$\forall t, t' \in T : \bullet t \cap \bullet t' = \emptyset$$

For EWFNs, conflict detection as discussed in Section 4.2.3 can be carried out based on the notion of *conflict free transitions* (cf. Definition 25). The existence of conflicts can be verified by checking if all transitions of an EWFN are conflict

free.

Definition 43 (conflict-free EWFN)**.** *An EWFN is called* conflict-free *iff*

$$\forall t \in T : t \textit{ is a conflict free transition}$$

If there are no conflicts in the model to be executed, the `sync` implementation can be reduced to a sequence of `take` or `read` operations since none of the operations overlap.

Recall that the Linda tuplespace model however includes very basic means for conflict resolution already: if two clients issue take operations that match the exact same tuple in the tuplespace, one of the clients is chosen non-deterministically to be the one that receives the tuple, while the other client is blocked until another match is available [Gel85]. Based on this special capability of tuplespaces, we can define a notion of weak conflicts. Weak conflicts are conflicts of transitions that involve one place. Based on Linda semantics, these situations can still be executed without the need for the mechanisms around set G and `SelectAndWakeUp`.

Definition 44 (weak conflict PT-net)**.** *A PT-net is called a* weak conflict *PT-net iff the PT-net is* not *conflict-free and*

$$\forall t \in T : \left|\{p|\ p \in \bullet t, \left|p\bullet\right| > 1\}\right| < 2$$

Weak conflict PT-nets contain transitions that consume from at most 1 place that is shared with another transition. Before we define this property for EWFNs, we first introduce the notion of a *conflict place* that is subsequently used to define *weak conflict* EWFNs.

Definition 45 (conflict place)**.** *A place $p \in P$ of an EWFN is called a* conflict place *iff*

$$\forall t, t' \in p\bullet : A(p,t) \dot{\cap} A(p,t') \neq \epsilon \wedge \\ \left(\langle p, t, \text{“take”}\rangle \in F \vee \langle p, t', \text{“take”}\rangle \in F\right)$$

A place p is called a *conflict place* if the intersection of the templates from two different transitions is not empty and one of the transitions issues a `take` operation.

Definition 46 (weak conflict EWFN)**.** *An EWFN is called a* weak conflict *EWFN iff the EWFN is* not *conflict-free and*

$$\forall t \in T : \left|\{p|\ p \in \bullet t, p \text{ is a conflict place}\}\right| < 2$$

A weak conflict EWFN contains only transitions that have at most 1 conflict place in their pre-set. It is important to note that all properties discussed can be statically analyzed (similar to the detection of conflict free transitions in Section 4.2.3), since their definition is not based on markings, but rather on static properties of the net.

To summarize: if a particular net to be executed is conflict free, no care has to be taken to resolve deadlocks or conflict situations, since both cannot happen. In this case, the net is structured in a way that no transition will ever be in conflict with another transition. If the net to be executed is weak conflict free, Linda semantics for conflict resolution are enough. The net is structured in a way that a transition is in conflict with another transition based on at most one place. This situation can be resolved by the tuplespace middleware by non-deterministically choosing one of the conflicting transitions to proceed. If the net to be executed is neither conflict free nor a weak conflict net, then the structure of the net contains potential for conflicts over 2 or more places and thus also for deadlocks as discussed in Section 6.4.1. Here, the mechanism around set G to suspend conflicting operations and to choose one operation to resolve the conflict is required.

6.4.3 Extended Tuple Matching Mechanism

The `sync` algorithm implements *join-matching* (recall its formalization in Section 4.2.4), an extension to the identity tuple matching used by most tuplespace implementations. Join-matching is used in `sync` operations to be able to enforce that certain fields in tuples matched by different templates of a `sync`

operation are equal. A use case for join-matching is to join two threads of control flow from the same process model and instance. The `sync` operation must only return if tuples with equal ProcessID and InstanceID are available in the tuplespace. We express this restriction by defining function $\texttt{check}_{\texttt{t}}$ (see Table 6.3) to be able to process templates of the form:

$$t_1 = \langle \text{“}CF\text{”}, ?\text{pid}, ?\text{iid} \rangle$$
$$t_2 = \langle \text{“}CF\text{”}, ?\text{pid}, ?\text{iid} \rangle$$

`?pid` and `?iid` refer to tuple elements on the second and third position. The `sync` operation returns only if the tuplespace contains tuples with identical values on the position of these variables.

Algorithm 6.17 shows an implementation of the $\texttt{check}_{\texttt{t,jv}}$ function that handles the matching for join-variables. It is a very close implementation of the term in Definition 26: for the `readall` operation at position (2), the join-variables are treated as wildcards (see the regular expression at position (1)). Then, an intermediate set U^{MS} is created which contains all matching tuples of the modified templates. The set U^{MS} of resulting tuples is checked for each join-variable if enough matching and equal values are available (3). This is done at position (4) by counting the number of occurrences of each value in the set tmp (using function `count`); if the biggest number of occurrences is greater or equal to the number of templates used for the `sync` operation, the particular join-variable is fulfilled since enough values fulfilling each individual template are available.

6.4.4 The `sync` Pattern

So far, we have presented a solution to efficiently synchronize on tuples residing in a *single* tuplespace. In decentralized EWFN execution however, tokens may be distributed over multiple tuplespaces and clients may need to synchronize over these distributed tuples. Figure 6.11 depicts this situation. The distributed version of the `sync` operation introduces additional challenges: in order to be able to process join-variables as described in Section 6.4.3, the data addressed by join-variables on all tuples has to be replicated between all tuplespaces that

Algorithm 6.17 `check`$_{\mathtt{t,jv}}$ Operation: Tuple Matching with *Join-Variables*

Require: $L \subseteq X$, set of templates with equal size (equal number of elements) for synchronized consumption. Note that the templates may contain join variables. $\exists n : \forall\ \mathrm{te} \in \mathrm{Te} : |\mathrm{te}| = n$.

Require: U^{MS}, temporary multiset used to buffer the results of the READALL operation.

Require: $T \subseteq X$, temporary set of templates modified by the regular expression.

```
function check_{t,jv}(L ⊆ X)
    T, U^MS ← ∅
    ∀te ∈ L : T ← T ∪ te[s/\?{1}\w+ /\⋆ /g]                    (1)
    ∀t ∈ T : U^MS ← U^MS + READALL(t)                           (2)
    if |U^MS| < |L| then return false end if

    for i ← 1 to |te|, te ∈ L do                                (3)
        if π_i(te) is join-variable then
            tmp ← ∅
            for all tu ∈ U^MS do
                tmp ← tmp ∪ π_i(tu)
            end for
            if count(tmp) < |L| then                            (4)
                return false
            end if
        end if
    end for

    return true
end function
```

are part of the `sync` operation. Only in this case, the algorithm is able to decide whether the operation can be unblocked or not. This introduces considerable network overhead, not only for the replication itself but also for keeping the replicated data up-to-date.

We present the *sync-pattern* that avoids the aforementioned problems by introducing an additional place (called *sync-place*, depicted by the shaded place

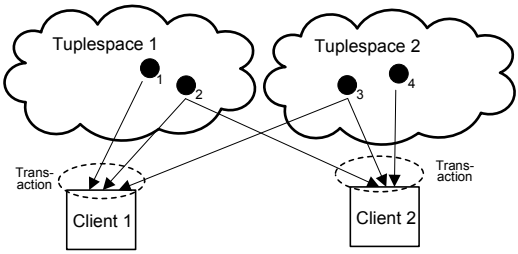
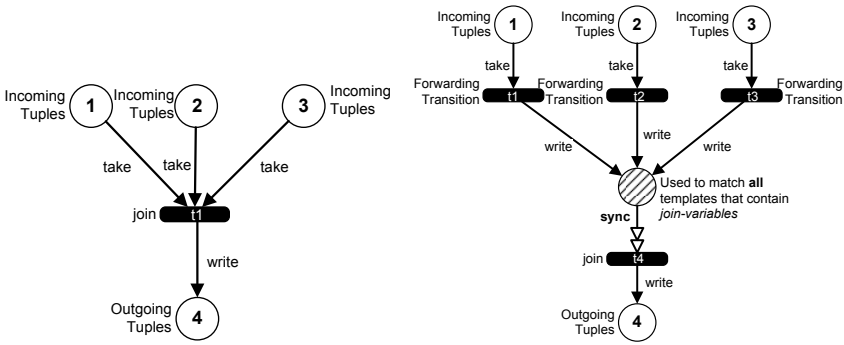


Figure 6.11: `sync` Operation over Tuples from Different Tuplespaces

in Figure 6.12(b)) and forwarding transitions to forward involved tuples to the sync-place. As a result, the distributed `sync` operation is reduced to a local `sync` as depicted in Figure 6.12. Note that this figure uses the EWFN notation that was introduced in Chapter 4. Also, places are annotated with numbers so that equivalent places in the transformed pattern can be easily located. Further note that forwarding transitions are usually not necessary and only shown to emphasize the similarity between the figures. During EWFN deployment (i.e. facilitated by the partitioning algorithm), the transitions that write tuples to the places 1, 2 and 3 (not shown on the figure) can be reconfigured to write directly into the sync-place, avoiding the need for forwarding transitions.

Note that this concept of handling joins is fundamentally different to the concept of a "Join-Service" in OSIRIS [Sch04]: here, the data structure that contains process instance data is called the *whiteboard*. In case of multiple concurrent execution paths, the whiteboard is split and distributed amongst all parallel execution partners, i.e. there exists a copy of process instance data on each node that is executed in parallel. As a result, complex and expensive mechanisms to merge instance data again after splitting it up for parallel execution is required. OSIRIS thus introduced a separate middleware component, the so called "Join-Service", to merge and create a consistent version of the instance data after joining parallel execution paths.

Figure 6.12(b) shows that the joining transition executes a `sync` operation on the sync-place, consuming tuples that were originally distributed over three



(a) Distributed `sync`

(b) Semantically Equal Local `sync`

Figure 6.12: The *sync-pattern* Optimization of Distributed `sync` Operations

places. In contrast to the distributed case (depicted in Figure 6.12(a)), global knowledge is available in the sync-place so that conflict situations can be resolved using Algorithm 6.14. Existing distributed `sync` operations that use templates with join-variables should be transformed according to the sync-pattern, so that the operation can be performed on a single place.

Statical analysis over the EWFN can be used to detect distributed `sync` operations:

Definition 47 (distributed `sync`). *Given* $|\bullet t| > 1$*, a transition* t *is said to execute a distributed* `sync` *operation iff*

$$\exists p, p' \in \bullet t : \mathsf{partition}_{PT}(p) \neq \mathsf{partition}_{PT}(p')$$

The function $\mathsf{partition}_{PT}$ is defined in [Wut10] and returns the partition that was assigned to the place given as argument. Distributed `sync` operations can be transformed according to the sync-pattern into local joins. This circumvents the need for replication of join-variable values over multiple spaces or the need for distributed transactions to coordinate individual `take` or `read` operations,

leading to a far more efficient implementation.

A drawback of the combination of the proposed `sync` coordination primitive and the sync-pattern is that it can only be used with tuples that are always read destructively. This is the case for control flow tuples in our envisioned distributed workflow engine, but it is not feasible for variables. As a result, the algorithm that creates the partitions needs to take distributed `sync` operations into account, and decide if it can be reduced to a local `sync` using the sync-pattern. Otherwise, distributed transactions with `read` or `take` operations as implemented by Algorithm 6.15 are required.

Definition 48 (reducible distributed `sync`)**.** *Given* $|\bullet t| > 1$*, a transition* t *is said to execute a* reducible *distributed* `sync` *operation iff*

$$t \text{ executes a distributed } \texttt{sync} \text{ operation} \wedge \forall p \in \bullet t : \langle t, p, \text{“take”} \rangle \in F$$

If there is a `read` operation involved in the `sync` operation, the operation is called a *non-reducible* distributed `sync`.

Definition 49 (non-reducible distributed `sync`)**.** *Given* $|\bullet t| > 1$*, a transition* t *is said to execute a* non-reducible *distributed* `sync` *operation iff*

$$t \text{ executes a distributed } \texttt{sync} \text{ operation} \wedge \exists \langle t, p, \text{“read”} \rangle \in F, p \in P$$

To summarize: the sync-pattern is an optimization that may only be applied in certain cases. Its prerequisite is a *reducible distributed* `sync` operation. If this prerequisite cannot be met, the implementation must fall-back to distributed transactions and `read`/`take` operations as demonstrated in Algorithm 6.15.

6.5 Conclusion

In this chapter, we discussed how EWFNs can be executed using tuplespace middleware. We started by analyzing different tuple structures to encode and execute two major kinds of Petri nets: Place-Transition and Colored Petri Nets. For both, we provided algorithms for their execution using JavaSpaces as the

tuplespace implementation, and used the basic tuplespace operations `read`, `write` and `take` to make the algorithms as generic as possible.

We then used the knowledge how to execute Petri nets to show how EWFNs – a special kind of Colored Petri Nets – can be executed. EWFN execution was further detailed by providing algorithms for the tuplespace operation equivalents of the different EWFN arcs. This included operations for the basic arcs `read`, `write` and `take`, but also for all extended arcs, namely `readall`, `takeall` and `update`.

Special attention was paid to the case of synchronized tuple consumption using the `sync` operation, and the potential presence of conflicts. We proposed an algorithm implementing the `sync` operation using JavaSpaces, but found it to be very inefficient. A solution was found by realizing the `sync` operation directly in the middleware, i.e. as part of the coordination interface of the tuplespace. This way, an efficient implementation of conflict resolution strategies was possible, since all active template operations on the tuplespace are known by the middleware.

In the following Chapter, we present the architecture and implementation of the tuplespace kernel that implements the presented operations, together with an evaluation of this system.

CHAPTER 7

ARCHITECTURE AND IMPLEMENTATION

In this chapter, we present the architecture as well as the implementation of prototype applications that implement the algorithms and techniques discussed in previous chapters. The result are two systems: (i) the coordination kernel as the basis for the decentralized workflow system implements the core of a tuplespace that was designed to efficiently execute EWFNs. (ii) A BPEL-to-EWFN compiler that transforms a given WS-BPEL 2.0 process model into an EWFN for partitioning and execution by the decentralized workflow system.

As a result, a tuplespace system is the only type of middleware underlying our workflow engine. Interestingly, using only one generic kind of middleware to build a workflow system has been suggested before: in [LR98], the authors discuss the architecture of IBM's FlowMark workflow system, which is based on MOM and RDBMS, i.e. it makes extensive use of persistent queues and stored procedures. The authors however suggest that the database system could be removed entirely. The resulting architecture was demonstrated by Alonso et al. in a research prototype named "FlowMark on Message Queue

Manager" (Exotica/FMQM, [AMG+95]). Here, FlowMark is re-architected to a distributed workflow system based purely on message-oriented middleware, a class of middleware which is similar to tuplespaces (cf. Chapter 3). In this chapter, we discuss the architecture and implementation of the tuplespace system that builds the groundwork for the "Process Space", our decentralized workflow engine.

We begin by presenting the architecture of the tuplespace kernel in Section 7.1 and how it integrates into the overall architecture of the "Process Space", the tuplespace system specifically designed to execute EWFNs. Subsequently, in Section 7.2 we present the coordination interface – the interface used by clients to coordinate their work – of our tuplespace system. This is followed by an evaluation of our prototype implementation in Section 7.3 that proofs the efficiency of our system and especially the `sync` operation, compared to the popular Blitz[1] JavaSpaces implementation.

Section 7.4 proposes EWFN-ML, an XML-based serialization format for EWFNs that is based on PNML[2], the ISO/IEC standard XML serialization format for Petri nets. This allows EWFNs not only to be used by the process partitioning algorithm defined in [Wut10], but by virtually any Petri net tool that accepts PNML as input format.

Finally, we present the design of the BPEL-to-EWFN compiler in Section 7.5 together with a brief example that shows a BPEL process and its resulting EWFN in graphical form.

7.1 Coordination Kernel Architecture

Figure 7.1 presents the high-level architecture of the "Process Space" prototype implementation, a tuplespace system that is designed around the algorithms and operations presented in Chapter 6 to execute EWFNs. In this section, we concentrate on the "kernel" of the system, i.e. how each of the coordination operations is implemented and how the resulting coordination interface looks like. All other components – such as functionality for management, deploy-

[1] `http://www.dancres.org/blitz/`
[2] `http://www.pnml.org/`

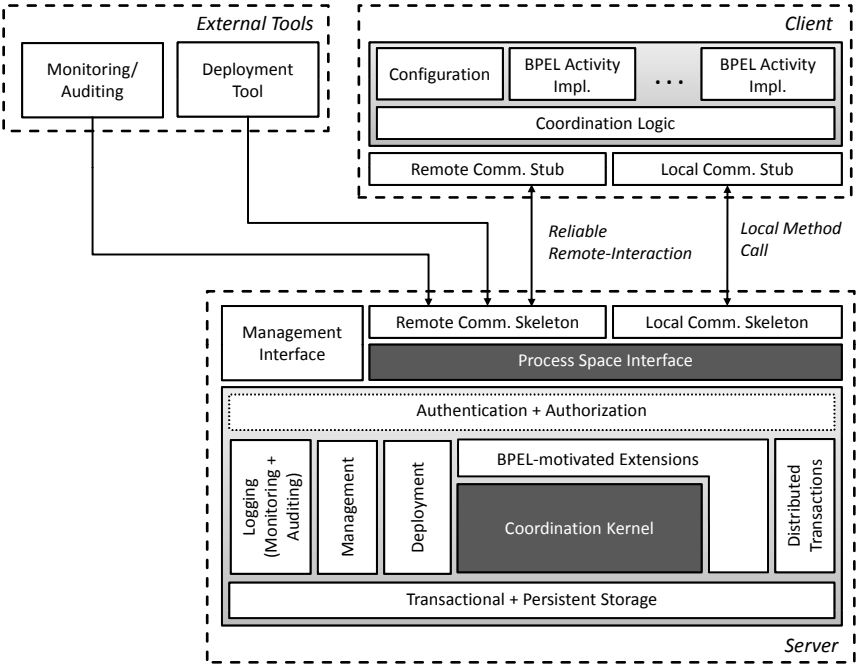


Figure 7.1: Process Space Architecture Overview [Wut10]

ment, security, transactions, etc. – are described and developed in the course of [Wut10]. The components making up the kernel of the Process Space system are highlighted in Figure 7.1 in dark color. The "coordination kernel"[1] component implements the algorithms and operations discussed in Chapter 6 (e.g. `read`, `write`, `take`, `sync`, ...) and interacts with the storage layer to persist the contents of the tuplespace. The "Process Space Interface" component exposes the coordination interface of the tuplespace system and is discussed in the next section (Section 7.2).

The implementation of the coordination kernel is based on the work carried

[1]Note that the term "coordination kernel" was also used in [Küh94]. There it refers to a different system that is not related to our project.

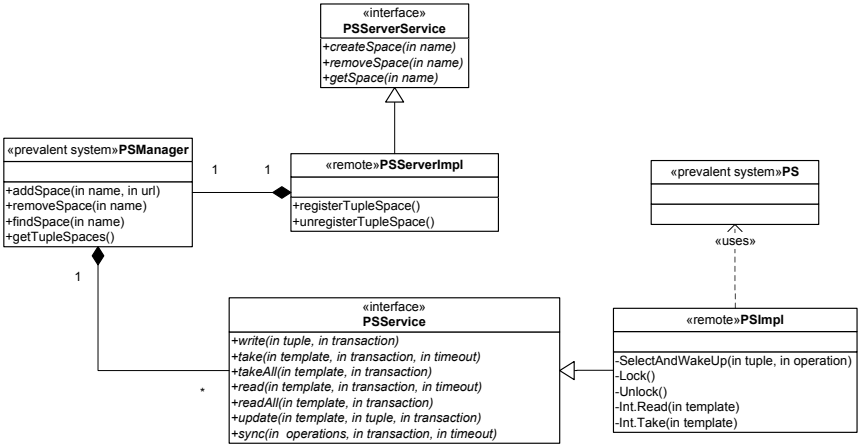


Figure 7.2: Simplified UML Class Diagram for the Process Space Kernel Implementation

out in [Wu08], where the first version of the prototype has been implemented. Figure 7.2 shows an UML Class Diagram [RBP+91] of how the final version of the kernel is implemented. Note that this figure is simplified for better readability and thus only shows the most important parts. Interface `PSServerService` and its corresponding implementation by class `PSServerImpl` form the entry point for a client application. Their functionality is thus exported through Java RMI[1], the remoting technology used in our prototype. They aggregate exactly one instance of the `PSManager` that provides methods to maintain the list of available tuplespaces. The `PSManager` again aggregates one or more instances of class `PSImpl` via the `PSService` interface, the central class of the kernel. It exports the coordination interface via RMI to the client and uses class `PS` which has two main purposes: (i) it is the *prevalent system*[2], i.e. it holds the object representation of the tuplespace to be stored by the transactional storage

[1] `http://java.sun.com/javase/technologies/core/basic/rmi/`

[2] A term used in the library we use to implement the transactional storage of our tuplespace implementation.

back-end[1]. (ii) It implements the algorithms presented in Chapter 6.

7.2 Coordination Interface

The *coordination interface* of a tuplespace denotes the interface that is visible to client applications that coordinate the tasks between each other by means of writing and (blocking) retrieval of tuples. In the following, we present the coordination interface of the "Process Space", our tuplespace implementation that clients use to coordinate the execution of a given process model. Our middleware is implemented in Java[2], the interface thus is defined in terms of a Java interface class (cf. class `TupleSpace` in Figure 7.2).

```
public TupleEntry read(TupleEntry template, Transaction
transaction, long timeout)
```

The `read` operation allows clients to non-destructively read tuples from the tuplespace. It is a blocking operation. Unless the timeout is set to 0, it blocks the client's thread for the amount of time given in the timeout parameter if no matching tuple is found. If more than one tuple matches the given template, it is not determined which tuple of the set of matching tuples is returned. The EWFN arc that corresponds to this operation is the `read` arc, discussed in Section 4.2.1. The operation is implemented according to Algorithm 6.7.

```
public Collection<TupleEntry> readall(TupleEntry template,
Transaction transaction)
```

The `readall` operation is a version of `read` that allows for batch consumption of all tuples matching a certain template, i.e. in contrast to `read`, it returns *all* matching tuples for a given template. The corresponding EWFN arc of the `readall` operation is the `readall` arc defined in Section 4.5.1; Algorithm 6.9 defines its implementation.

[1] Implemented by Prevayler (`http://www.prevayler.org/`).
[2] `http://www.java.com/`

```
public TupleEntry take(TupleEntry template, Transaction
transaction, long timeout)
```

The `take` operation is the destructive version of `read`, i.e. it removes the tuple that matches the given template from the tuplespace instead of returning a copy. Similar to `read`, it is not determined which tuple of a set of matching tuples is returned. Moreover, if two or more operations compete for the same tuple (a.k.a. a conflict), it is not determined which operation of the list of competing operations is selected to return the tuple. Similar to `read`, `take` is a blocking operation if parameter timeout has a value greater than 0. The `take` operation is implemented by Algorithm 6.10, its corresponding arc type in EWFNs is the `take` arc which is defined in Section 4.2.1.

```
public Collection<TupleEntry> takeall(TupleEntry template,
Transaction transaction)
```

Similar to `readall`, the `takeall` operation is the batch version of `take` that returns `all` matching tuples instead of choosing just one match. All matching tuples are removed from the tuplespace. The `takeall` operation is implemented according to Algorithm 6.12, its corresponding EWFN arc is described in Section 4.5.2.

```
public void update(TupleEntry template, TupleEntry value,
Transaction transaction)
```

To enable in-place updates of tuples without the need for clients to consume them, update them on the client side and write them back to the tuplespace, the `update` operation has been added. It is an optimization that avoids unnecessary data transfer between the client and the tuplespace system. Furthermore, it is an atomic operation in contrast to a (non-atomic, if no transaction is used) sequence of `read` and `take` as it would be required to update a tuple. The implementation of the `update` operation is based on Algorithm 6.13, its corresponding EWFN arc is described in Section 4.5.3.

```
public Collection<TupleEntry> sync(Collection
<ReadAndTakeEntry> operations, Transaction transaction, long
timeout)
```

The `sync` operation – as discussed in Section 6.4 – allows for consuming sets of tokens from a tuplespace in a blocking manner, employing mechanisms for conflict resolution and prevention of deadlocks. A timeout of 0 means that the operation does not block and returns immediately. In our implementation, the `sync` operation works according to Algorithm 6.16, its corresponding EWFN arc is described in Section 5.1.1.1.

```
public void write(TupleEntry tuple, Transaction transaction)
```

The `write` operation adds the tuple given as parameter to the tuplespace. It is implemented according to Algorithm 6.6 and implements the EWFN arcs going from transitions to places as documented in Section 4.2.1.

Note that all operations of the coordination interface support the (optional) transaction parameter. It allows to group multiple operations, including operations on other (possibly remote) tuplespaces, into a single unit of work coordinated via a transaction. If multiple tuplespaces are involved, the (then) distributed transaction is carried out using the Two-Phase-Commit (2PC) protocol [GR93]. Using a `null` value for this parameter means that the effects of the corresponding operation should be made visible immediately[1].

All tuple consumption operations (`read`, `take` and `update`) allow for the definition of XPath queries to match tuples containing fields with XML content. This avoids the need to transfer big chunks of data if only small parts of a big document are needed. This way, reading or updating a single value from a big XML document does not require to transfer the whole document to the client and apply the XPath statement locally. Instead, the statement is applied by the middleware and only the result is returned to the client[2].

[1] Similar to `autocommit` in JDBC for instance.

[2] This is similar to stored procedures in a traditional RDBMS or predicate push-down in federated database systems.

7.3 Evaluation

In this section, we evaluate the functionality of our tuplespace prototype when executing Petri nets. We compare our execution results to a similar implementation that uses Blitz JavaSpaces instead of our own tuplespace prototype as underlying middleware.

7.3.1 Evaluation Process

To validate our approach of tuplespace-based Petri net execution, we evaluate our system along two different dimensions: (i) we use both Petri net encodings (model 1 from Section 6.1.2 and model 2 from Section 6.1.3) presented in Chapter 6 and thus execute a CP-net directly as well as its equivalent PT-net, and (ii) we evaluate different underlying tuplespace implementations for the algorithms of model 1 and model 2.

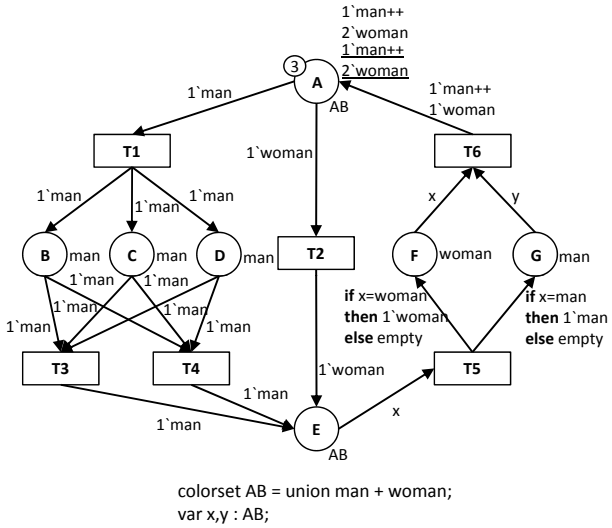


Figure 7.3: Colored Petri Net Used for the Evaluation [MWL09]

Figure 7.3 presents the process used for our evaluation. It is modeled as

Colored Petri Net[1] using the syntax proposed by [Jen92], i.e. each arc is annotated with an expression that evaluates to (or directly is) a tuple; *empty* means that no tuple is communicated. Each place is annotated with a color set (*AB*, *man* and *woman* in our case) denoting the type of tokens it may contain. The multiset of tokens the initialization function for each place evaluates to is indicated by an underlined expression. The current marking of a place is denoted by a small circle around the number in the multiset expression.

In the CP-net depicted by Figure 7.3, the initial marking is one "man" token and two "woman" tokens in place A. This enables both, transitions t_1 and t_2 which produce three "man" and one "woman" token respectively upon firing. Subsequently, the transitions t_3 and t_4 are both enabled, but in conflict, since both have the exact same enablement condition (both consume exactly one "man" token from the places B, C and D). Place E merges[2] both execution paths. Transition t_5 is enabled in both two: a "man" or a "woman" token in place E. Transition t_5 models a choice; it either produces a woman token on place F or a "man" token on place G. Transition t_6 synchronizes "man" and "women" tokens (it fires only if at least one "man" and one "woman" token is present in each of its input places). Upon firing, it produces a "man" and a "woman" token on place A which restarts the cycle.

Since we want to execute workflow definitions in Petri net form, the process we use for evaluation is constructed from the six basic workflow control flow patterns documented in [RtHvdAM06]:

1. WCP-1 (Sequence) is implemented by the path $A \rightarrow T2 \rightarrow E$
2. WCP-2 (Parallel Split) is implemented by the path $T1 \rightarrow B, C, D$
3. WCP-3 (Synchronization) is implemented by the path $F, G \rightarrow T6$
4. WCP-4 (Exclusive Choice) is implemented by the path $T5 \rightarrow F, G$
5. WCP-5 (Simple Merge) is implemented by the path $T3, T4 \rightarrow E$
6. WCP-16 (Deferred Choice) is implemented by the path $A \rightarrow T1, T2$

In Figure 7.4 we present the resulting, semantically equivalent TP-net from the transformation of Figure 7.3 by following the construction rules given in

[1] Using CPN Tools v2.2.0, `http://wiki.daimi.au.dk/cpntools/cpntools.wiki`
[2] but does not synchronize

the proof of the equivalence of PT-nets and non-hierarchical CP-nets [Jen92]. The resulting PT-net is very similar to the original CP-net since the number of colors is very low (in fact, we use only two colors, *man* and *woman*) so that the resulting PT-net is still easy to understand and the semantic equivalence can be verified by playing the "token game" with both nets. The initial marking is depicted by black tokens in the places $\langle A, m\rangle$ and $\langle A, w\rangle$.

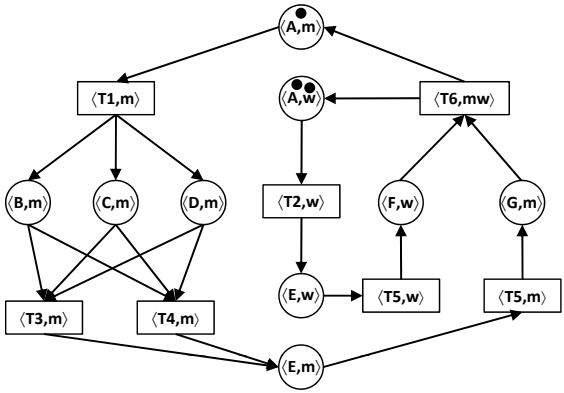


Figure 7.4: Semantically Equivalent Place-Transition Net [MWL09]

Places and transitions are labeled by tuples $\langle p, c\rangle \in PE, p \in P, c \in C(p)$ (PE is called the set of *place elements*) and $\langle t, b\rangle \in BE, t \in T, b \in B(t)$ (BE is called the set of *binding elements*) respectively. In the figure, we use m and w as shorthand notations for the colors *man* and *woman* from the original CP-net. Note that the sets PE, BE, $C(p)$, P, T and $B(t)$ are part of the original definition of CP-nets [Jen92].

7.3.2 Discussion of Results

We implemented two applications to execute Petri nets based on the algorithms for the Petri net encoding models 1 and 2 proposed in Chapter 6. One uses *Blitz JavaSpaces* as the underlying JavaSpaces implementation, the other uses our new tuplespace prototype implementation (see Section 7.1). We used these two implementations to evaluate our different encodings as well as

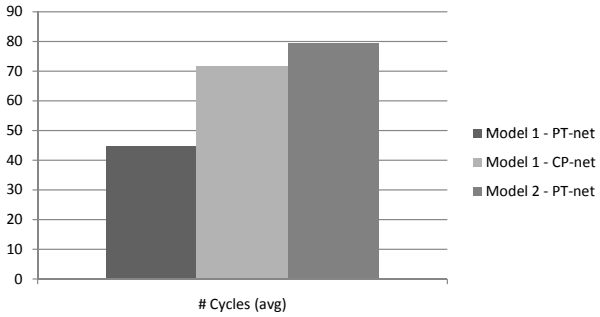
our tuplespace implementation. Consequently, both applications have three modes of operation (called "test-cases" later on): (i) CP-net execution using the algorithms of model 1, (ii) PT-net execution using the algorithms of model 1, and (iii) PT-net execution using the algorithms of model 2. Note that the combination CP-net execution using model 2 is missing because model 2 cannot represent colored tokens.

Before starting each application, we changed the configuration parameters of the underlying tuplespace systems in order to provide a similar-sized disk write buffer. This buffer keeps pending write operations in memory and writes all of them to disk at once when it is full. Both applications were allowed to keep 100 operations in memory, i.e. a maximum of 100 write operations would be lost in case of system failure. We also disabled acknowledgment messages for remote calls on the Process Space prototype in order to provide at most once semantics, i.e. equal to what Blitz JavaSpaces provides by using plain RMI.

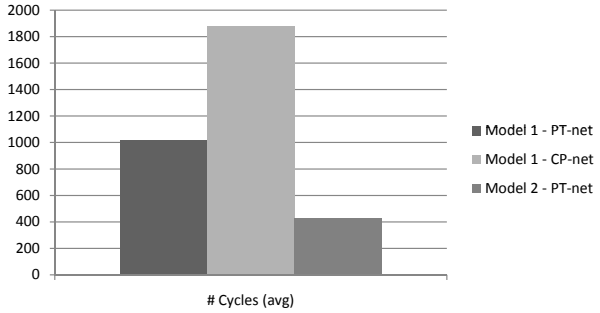
For the evaluation, we benchmark both applications based on the process modeled by the CP-net from Figure 7.3 and its PT-net equivalent shown in Figure 7.4. The goal of the benchmark is to measure the average *throughput* of basic workflow control flow patterns. The process to be executed is cyclic, i.e. its execution results in an endless loop. In both Petri nets, transition T6 that joins the alternative paths and starts a new iteration of the loop is used as counter. Each time T6 fires, the cycle counter for the executed Petri net is increased by one. The benchmark then measures how many cycles each of the applications can achieve for a specific Petri net in a given amount of time.

In total, six different tests were carried out by testing both applications based on three test-cases: each application executed the PT-net using model 1 and 2, and the CP-net using model 1. The tests were conducted on a Windows XP machine with an Intel Pentium M processor and 2GB of memory, running Sun's JDK 1.6.0_11. To minimize errors caused by side effects such as Java JIT warm-up, each test was run multiple times and the average over the result was calculated.

The average number of completed cycles within 30 seconds for both applications executing the three test-cases is shown in Figure 7.5. Higher numbers



(a) Average Number of Cycles for Each Encoding and Petri net Type with the JavaSpaces-based Application



(b) Average Number of Cycles for Each Encoding and Petri net Type with the Process Space-based Application

Figure 7.5: Average Number of Cycles Measured with Each Implementation [MWL09]

mean that more cycles could be completed and thus that better performance was achieved. In the best case, the application using the Process Space middleware completed over 22 times more cycles than the best case of the application using Blitz JavaSpaces. This is due to the `sync` operation employed by the Process Space-based application, which allows it not to rely on transactions as means to resolve conflicts between transitions that mutually consumed tuples. As a result, it does not suffer from the heavy performance impact caused by

transaction rollback and retry loops as seen in the Blitz JavaSpaces-based application. Moreover, it greatly simplifies the algorithm to implement transition semantics for tuplespace clients. While the Blitz JavaSpaces-based application uses the algorithms described in Section 6.1.2 and 6.1.3, the Process Space-based application does not require the employed loops with `take` operations, i.e. the implementation of incoming arcs. The two nested loops at position (3) and (6) in Algorithm 6.2 for instance are replaced by a single `sync` operation. Similarly, `sync` replaces the loops at positions (2) and (3) in Algorithm 6.4.

Figure 7.5 shows an interesting observation: independent of the underlying middleware, using model 1 always results in more completed cycles for CP-net execution than executing its PT-net equivalent. The reason is that the number of elements (transitions and places) of a CP-net generally is smaller than the number of elements of its (transformed) equivalent PT-net. As a result, a cycle in our benchmark CP-net includes fewer steps than in its PT-net version. Using the CP-net definition and the corresponding sets TE, $C(p)$, BE and $B(t)$ from [Jen92], the number of places and transitions generated by creating an equivalent PT-net can be estimated as:

$$\begin{array}{rclcl} \left|P_{\text{PT-net}}\right| & = & |TE| & = & \sum_{\forall p \in P} \left|C(p)\right| \\ \left|T_{\text{PT-net}}\right| & = & |BE| & = & \sum_{\forall t \in T} |B(t)| \end{array}$$

Generally, the number of elements in an equivalent PT-net depends on the number of colors defined in the given CP-net. We can thus give an upper bound for the number of elements by the following term:

$$|C| * (|P| + |T|)$$

Remarkably, execution using model 2 leads to the highest number of cycles on the JavaSpaces based system, whereas it shows by far the lowest number on the application based on the Process Space prototype. The reason for this is the non-blocking nature of the client library that exports RMI methods to offer callbacks to be notified when matching tuples are available (instead of constantly polling for matching tuples). At the high rates of operations

executed on the tuplespace, exporting and un-exporting RMI callback methods becomes the limiting factor for model 2; in contrast to model 1 it requires two (a `take` and a `write` operation) operations per incoming and outgoing arc as shown by the body of the `for` loops in Algorithm 6.4, instead of just one operation as shown in Algorithm 6.2.

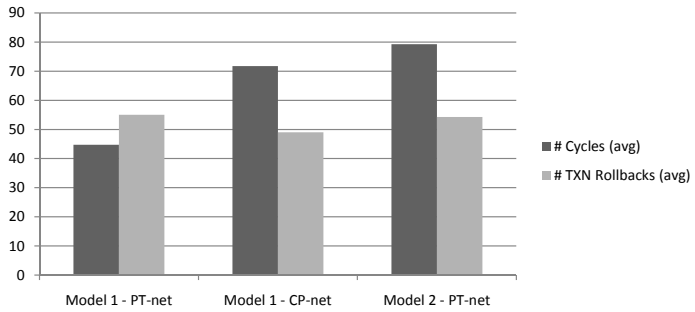


Figure 7.6: Average Number of Cycles vs. Average Number of Transaction Rollbacks in the JavaSpaces-based Application [MWL09]

The case of transaction rollbacks is emphasized by Figure 7.6. It proves that transaction rollbacks for conflict resolution are the limiting factor for the JavaSpaces-based application. In the case of PT-net execution using model 1, there were more rollbacks than completed cycles, i.e. on average there was more than one transaction rollback per cycle. This figure also confirms the observations made on Figure 7.5: the fact that the CP-net generally has fewer elements than an equivalent PT-net derived through transformation leads to (i) faster execution time of the CP-net when using the same encoding (model 1) as shown by the first two dark columns (counted from left to right) in Figure 7.6 and (2) slightly less transaction rollbacks since fewer elements (and thus fewer arcs with potential for conflicts) are involved in the process model.

In Section 6.1.5 where we discussed the qualitative evaluation of the Petri net encodings, we predicted that the coarser granularity of operations introduced by model 2 causes a higher likelihood of transaction rollback when compared to model 1. The measurements of model 1 and 2 for PT-nets execution however

do not support this claim: Figure 7.6 does not show significant differences in terms of transaction rollbacks between the encoding models. This is because the coarser granularity of operations in model 2 becomes only a problem when access to multiple tokens residing in a single place is blocked by a transition consuming a subset of the available tokens. In this case, the transition prevents other transitions from being enabled although the current marking of places would enable them. The reason for this is that the place tuple has to be consumed as a whole, the tuples that it contains are not available to other transitions until it is written back to the tuplespace. In the Petri nets used for our test-cases, this particular situation cannot occur and thus no significant difference was detected.

7.4 EWFN XML Representation (EWFN-ML)

EWFNs are the input for the partitioning algorithm that creates the actual partitions for decentralized execution. In this section we define EWFN-ML, the serialization format for EWFNs that is used by the partitioning algorithm. We define EWFN-ML on the basis of PNML [B+03], an ISO/IEC standardized[1] serialization format for Petri nets. EWFN-ML is valid PNML, it uses the extension elements of PNML to embed the Process Space specific details into the standard PNML syntax. As a result, EWFN-ML files can be imported into any Petri net tool that accepts PNML[2], e.g. to visualize (see Figure 7.16 for an example) a given EWFN. This connects EWFNs to the world of Petri net analysis algorithms and tools such as Tina [BRV04], all tools based on the "Petri Net Kernel" [KW01] or WoPeD[3].

[1] See ISO/IEC 15909 Part 2: `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43538`

[2] The EWFN specific PNML elements are ignored by these tools.

[3] `http://www.woped.org/`

7.4.1 The Petri Net Markup Language (PNML)

The PNML grammar is defined in RELAX NG[1] an OASIS standardized[2] XML grammar definition format similar to XML Schema [T+04]. To be compatible to the PNML grammar, we define the PNML extensions that define EWFN-ML in RELAX NG as well. In the following, we start with a brief example of PNML and then concentrate on PNML extensibility, specifically the `<toolspecific>` element.

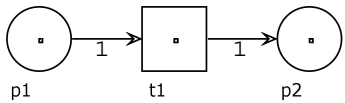


Figure 7.7: Simple Petri Net to be Serialized as PNML

Figure 7.7 shows a simple Petri net, its PNML serialization is presented in Listing 7.8. The Petri net consists of the places p1 and p2, transition t1 and two arcs that connect transition t1 with the two places. In PNML, each element is defined separately and "wired" via the `source` and `target` attributes of the `<arc>` element. Extensibility of this basic PNML structure is provided by the `<toolspecific>` element, an optional child XML element the PNML schema allows for `<page>`, `<place>`, `<transition>` and `<arc>`. In Listing 7.8 this concept is exemplified by the `<toolspecific>` element the WoPeD editor generates for transitions (see line 18). Here, it is used to embed tool-specific information regarding timing information of a transition for simulation purposes into the Petri net.

[1] http://www.relaxng.org/
[2] http://www.oasis-open.org/committees/relax-ng/

```
<pnml xmlns="http://www.pnml.org/version−2009/grammar/pnml">
  <net id="1" type="http://www.pnml.org/version−2009/grammar/ptnet">
    <page id="top−level">
      <place id="p1">
        <name>
          <text>p1</text>
        </name>
      </place>
      <place id="p2">
        <name>
          <text>p2</text>
        </name>
      </place>
      <transition id="t1">
        <name>
          <text>t1</text>
        </name>
        <toolspecific tool="WoPeD" version="1.0">
          <time>0</time>
          <timeUnit>1</timeUnit>
        </toolspecific>
      </transition>
      <arc id="a1" source="p1" target="t1">
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
      <arc id="a2" source="t1" target="p2">
        <inscription>
          <text>1</text>
        </inscription>
      </arc>
    </page>
  </net>
</pnml>
```

Figure 7.8: PNML Serialization of the Petri net Shown by Figure 7.7

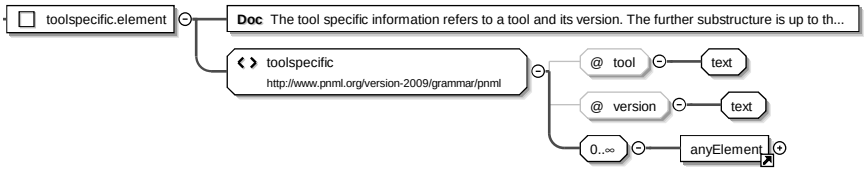


Figure 7.9: RELAX NG Schema for the `<toolspecific>` PNML Element

The <toolspecific> part of the PNML RELAX NG schema is depicted in Figure 7.9. This figure uses the Oxygen RELAX NG graphical notation, generated using the Oxygen RELAX NG schema editor[1]. It consists of the two mandatory attributes tool and version and zero or more XML elements that match anyElement.

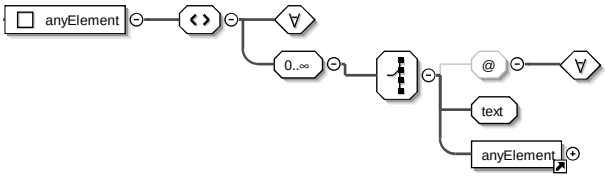


Figure 7.10: RELAX NG Schema for anyElement Referenced in Figure 7.9

The RELAX NG schema for anyElement is depicted in Figure 7.10. It recursively allows any XML element containing any number of attributes or text elements.

7.4.2 EWFN-ML Schema

Figure 7.11 shows the RELAX NG schema for the EWFN-ML specific part of the PNML <toolspecific> element. It contains one of four different elements, depending on the type of its parent PNML element. If the parent is a <page>, the <bpelprocess> element defines the namespace and the process name of the BPEL process the EWFN was generated from. When an <arc> is the parent element of <toolspecific>, the fields <operation>, <type> and <cause> define the Process Space operation, the type of the arc and the cause (the arc navigated before, that caused activation of the current arc). In case of a <transition> as parent element, <toolspecific> contains <ewfnelement> and <bpelelement>, defining the EWFN element name and the BPEL element name the transition implements respectively. In case of a <place> as parent element, <ewfnelement> defines the EWFN element name of the particular place.

[1] http://www.oxygenxml.com/relaxng_schema_editor.html

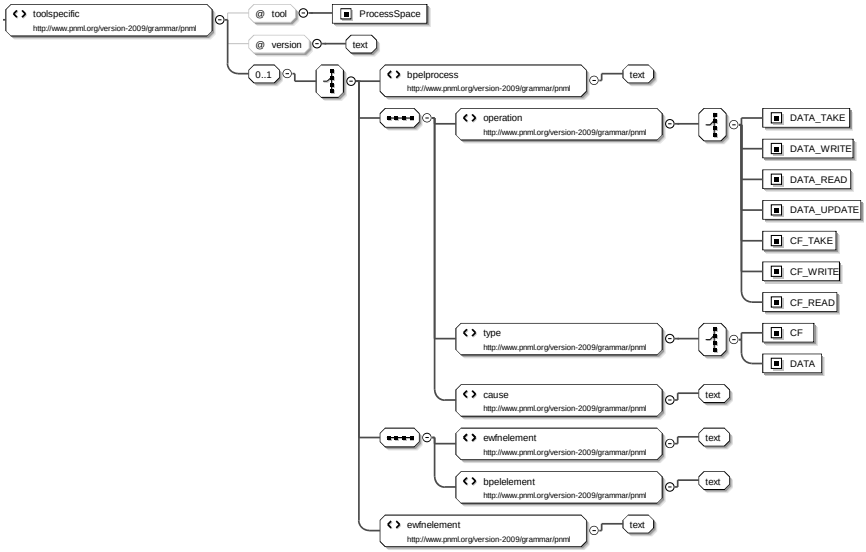


Figure 7.11: RELAX NG Schema for the `<toolspecific>` Element of EWFN-ML

7.4.3 EWFN-ML Example

Listing 7.12 shows EWFN-ML with `<toolspecific>` elements that match the EWFN-ML grammar from Figure 7.11. It is a minimal example that shows exactly one element from all possible EWFN-ML elements and contains example values for its fields.

```
<pnml xmlns="http://www.pnml.org/version−2009/grammar/pnml">
  <net id="n1" type="http://www.pnml.org/version−2009/grammar/ptnet">
    <page id="top−level">
      <toolspecific tool="processspace" version="1.0">
        <bpelprocess>http://example.com/loan−approval/loanApprovalProcess</bpelprocess>
      </toolspecific>

      <place id="P_1">
        <name>
          <text>var_myVar_root</text>
        </name>
        <toolspecific tool="processspace" version="1.0">
          <ewfnelement>variable</ewfnelement>
        </toolspecific>
      </place>

      <transition id="T_1">
        <name>
          <text>scope_t1</text>
        </name>
        <toolspecific tool="processspace" version="1.0">
          <bpelelement>HelloWorld−Process</bpelelement>
          <ewfnelement>SCOPE_START</ewfnelement>
        </toolspecific>
      </transition>

      <arc id="A_1" source="P_1" target="T_1">
        <toolspecific tool="processspace" version="1.0">
          <operation>CF_TAKE</operation>
          <type>CF</type>
          <cause>start</cause>
        </toolspecific>
      </arc>
    </page>
  </net>
</pnml>
```

Figure 7.12: EWFN-ML Example Showing the Possible Elements and their Contents

7.5 BPEL to EWFN-ML Transformation

In this section we present the architecture and implementation of the BPEL-to-EWFN compiler. This tool takes a BPEL file as input, generates an EWFN using the EWFN patterns discussed in Chapter 5 and serializes the result in EWFN-ML.

7.5.1 Architecture

Figure 7.13 shows the high-level architecture of the BPEL-to-EWFN compiler. It consists of four modules that use an XML-Parser to parse the .bpel input file and generate the EWFN in form of a .pnml file as output. Instead of creating our own BPEL parser and object model, we use a stripped-down version of these components from the Eclipse BPEL Designer[1] open-source project.

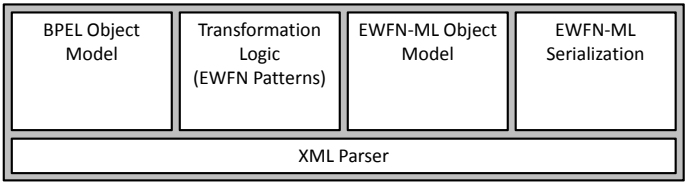


Figure 7.13: High-Level Architecture of the BPEL2EWFN Compiler

The transformation logic that populates the EWFN-ML object model from the BPEL object model is implemented according to the EWFN patterns documented in Chapter 5. The transformation operates at the level of composite transitions, i.e. the granularity has changed from single transitions to a composite transition per BPEL activity in order to reduce the number of transition implementations the decentralized execution engine needs to provide. Finally, the EWFN object model is serialized as EWFN-ML in form of a .pnml file.

The general data flow within the prototype application is depicted by Figure 7.14. The BPEL object model is the input for the BPEL-to-EWFN transformation logic that operates on the EWFN object model. After the transformation

[1] `http://www.eclipse.org/bpel/`

Figure 7.14: Dataflow Within the BPEL2EWFN Compiler

is done, the fully populated EWFN object model is serialized as EWFN-ML to a .pnml file.

7.5.2 Implementation

Figure 7.15 presents a simplified UML class diagram of the main classes from the BPEL-to-EWFN compiler. The upper part shows the EWFN object model representing transitions (class `EWFNMLTransition`) and places (class `EWFNMLPlace`) that are derived from the base class `EWFNMLVertex`. Class `EWFNMLArc` is derived from `DefaultWeightedEdge` that is part of the JGraphT package[1]. We use JGraphT as the basis for implementing the EWFN graph partitioning algorithm. An EWFN document is represented by the class `EWFNML` that aggregates the individual elements of the object model. Class `BPEL2EWFN` provides the main routine of the program and contains the implementation of the transformation logic that uses the BPEL object model and creates the EWFN-ML document. Serialization of the EWFN object model to PNML is done by calling the `toXML()` methods of all contained objects. The serialization logic thus is distributed over each class of the object model. The idea is that each class generates the PNML version of the state it contains. Class `EWFNML` then puts the individually generated EWFN-ML parts into the correct order to produce the serialization of the EWFN graph it represents.

The implementation of the compiler is designed to be extensible. The support for mapping another language into EWFNs can be added by providing alternative implementations for the first two components (counting from left to right) in Figure 7.13, i.e. an object model representing the constructs of the

[1] `http://www.jgrapht.org/`

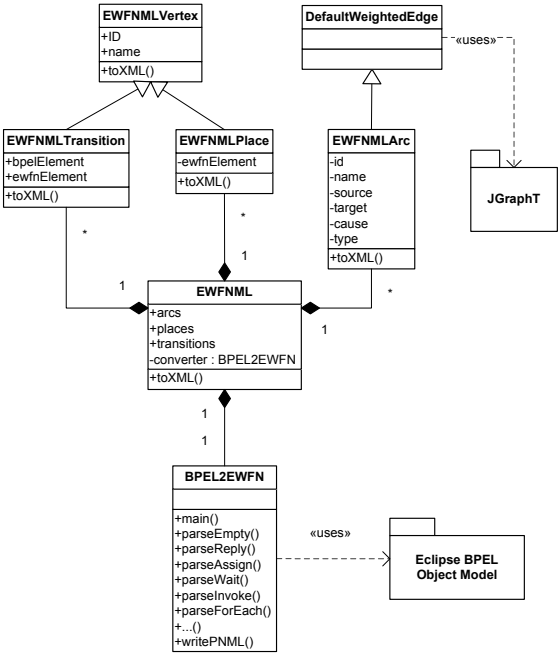


Figure 7.15: UML Class Diagram of the BPEL2EWFN Compiler

new language and a corresponding implementation of class `BPEL2EWFN` from Figure 7.15 that contains the transformation rules, must be implemented.

7.5.3 Transformation Example

The BPEL-to-EWFN compiler has a simple command-line interface with the syntax:

```
bpel2ewfn.sh [BPEL-File] [EWFNML-File]
```

Invoking the following command with file “loanapproval.bpel” in the current directory generates a file named `loanapproval.pnml` that contains the EWFN-equivalent of `loanapproval.bpel`.

```
./bpel2ewfn.sh loanapproval.bpel loanapproval.pnml
```

As the generated EWFN-ML is PNML compliant XML, the generated file may not only be used by the partitioning algorithm developed in [Wut10], but also by any Petri net tool that imports .pnml files. For debugging purposes, we implemented a tool based on the Petri Net Kernel [KW01] that visualizes EWFNs. Figure 7.16 shows the output of this tool when fed with the EWFN which was generated by the BPEL-to-EWFN compiler from a simple BPEL process with a `sequence` of `receive`, `assign` and `reply`[1].

[1] Note that this figure is included as a reference to see how an actual EWFN may look like.

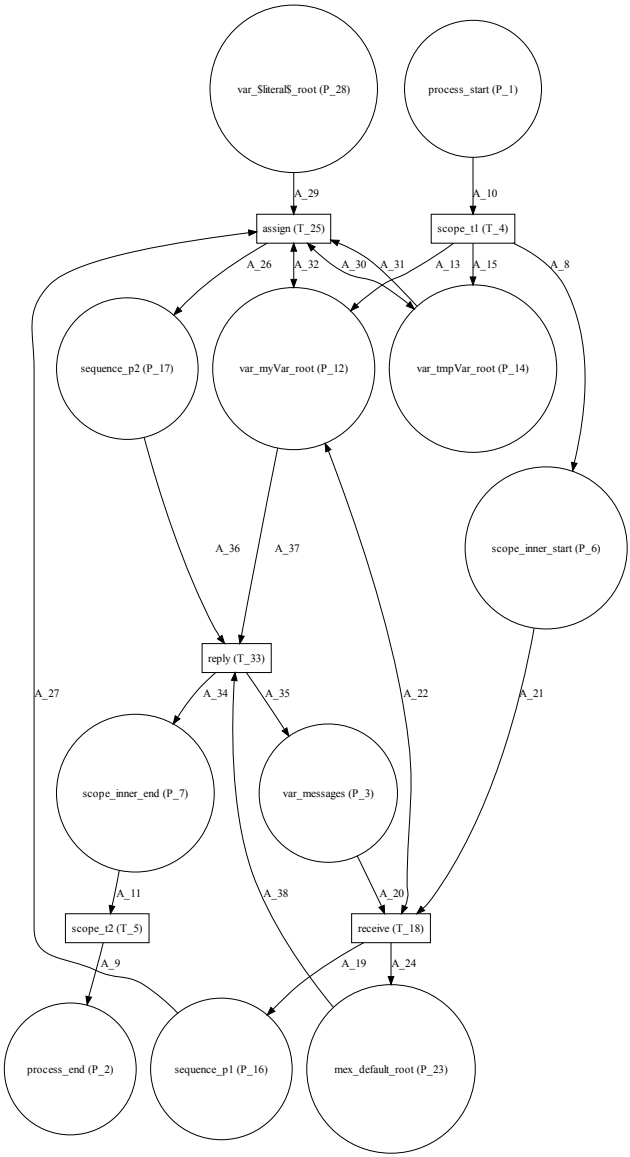


Figure 7.16: A Simple BPEL Process with a `sequence` of `receive`, `assign` and `reply` in EWFN Form

7.6 Conclusion

This chapter presented the architecture and implementation of prototype applications that realize the concepts developed in the previous chapters. We started by discussing the implementation of the tuplespace kernel that is the basis for our decentralized workflow execution engine. The coordination interface of the tuplespace kernel was specifically designed to execute EWFNs, the execution model higher-level workflow languages like BPEL are translated to in order to be executed. Subsequently, we evaluated the tuplespace kernel prototype along two dimensions: (i) we used the two different encoding models developed in Chapter 6 to execute the same Petri net in order to find out strengths and weaknesses of each encoding, and (ii) we evaluated different tuplespace implementations for the execution of these encoding models. The results proved the efficiency of the `sync` operation and the conflict resolution strategies it implements. Next, we defined EWFN-ML, the serialization format of EWFNs. EWFN-ML is based on PNML, an ISO/IEC standard for representing Petri nets in XML form. In fact, EWFN-ML is valid PNML since all EWFN specific elements are expressed using the extension mechanism of PNML. As a result, EWFN-ML is not only used by the partitioning algorithm of our workflow engine implementation, but can also be imported into any Petri net tool that understands PNML, e.g. for visualization or statical analysis. Finally, we presented the architecture and implementation of the BPEL-to-EWFN compiler that generates EWFN-ML from a given WS-BPEL process according to the EWFN patterns presented in Chapter 5.

CHAPTER 8

CONCLUSION AND OUTLOOK

This work provided the theoretical underpinnings and the core parts of the implementation of the "Process Space", a tuplespace-based execution engine that allows for fully decentralized execution of workflows. Based on the paradigm of *token passing*, the engine innovatively combines its two technological underpinnings: *tuplespaces* and *Petri nets*. Tuplespaces provide a suitable middleware for the implementation of the process engine and Petri nets offer a matching theory for the formalization of tuplespace-based interactions. Moreover, Petri nets are a widely accepted and used theory in workflow research, and thus match our requirements to map a high-level process execution language such as WS-BPEL into our new execution model.

Before defining the execution model, we investigated existing workflow engines and their architectures in Chapter 3 in order to learn how these systems implement the "navigator", i.e. the core component of a workflow management system that is responsible for governing the execution of the activities a process is composed of. This component is typically implemented using message-oriented middleware and RDBMS. Our analysis concluded that tuplespaces – as the basis for the implementation of our decentralized navigator – are a viable alternative.

The knowledge gained from this analysis provided the basis for the development of the decentralized execution model, called "Executable Workflow Nets" (EWFN) which is defined in Chapter 4. We provided a formal description of the syntax and semantics of EWFNs together with a corresponding graphical representation. Moreover, we developed a mapping from $\text{EWFN}_{\text{Linda}}$ to Place-Transition Nets, allowing standard Petri net analysis and verification techniques to be used with $\text{EWFN}_{\text{Linda}}$. EWFNs are called "directly executable" on tuplespace middleware, since each element of their model has an equivalent element in a tuplespace-based application. As a result, EWFNs are a generic formalism and graphical notation to model tuplespace-based applications.

Subsequently, in Chapter 5 we used EWFNs to express the semantics of WS-BPEL 2.0 and thus mapped a largely used workflow definition language into our decentralized execution model. By means of EWFNs, the semantics of each BPEL activity is specified in terms of tuplespace operations and thus in a way suitable for our decentralized execution engine. We covered BPEL's basic as well as structured activities and the `scope`, an activity with semantics that have proven to be especially complex to maintain in distributed settings.

Next, in Chapter 6 we investigated different possibilities to execute EWFNs using tuplespaces. We started by introducing generic methods to execute different kinds of Petri nets such as Place-Transition or Colored Petri Nets. Each of these Petri net "dialects" requires a different way to encode the structure of the net in tuple form. EWFNs are based on Colored Petri Nets, we thus can use the encoding strategy for Colored Petri Nets for EWFNs as well. We subsequently presented algorithms on tuplespaces to implement basic EWFN arcs such as `write`, `read`, `take`, `readall` and `takeall`. Furthermore, we discussed the problem of synchronized consumption of tuples in tuplespaces and presented a solution in form of the `sync` coordination primitive.

The concepts presented in this thesis are realized with the implementation presented in Chapter 7. We discussed the architecture and implementation of the tuplespace kernel that natively executes EWFNs and thus builds the basis for the proposed, decentralized workflow engine. In particular, the presented tuplespace kernel implements the tuplespace operations discussed in the previous chapter and defines EWFN-ML, the XML-based serialization

format of EWFNs.

8.1 Future Work

The results of this work may be used as the underpinning for various directions in future research and developments:

Customizations for scientific computing / simulation scenarios Together with the Cluster of Excellence "Simulation Technology" (SimTech)[1] at the University of Stuttgart, we are working on customizations of our workflow engine for applications in the area of scientific simulations [SGK+10]. The use case for our engine is a specialization of scenario 3 presented in Section 1.3. Scientific simulations typically involve shipping large amounts of data between activities, our engine in this case avoids the need to ship the data from and to a central WfMS, but instead allows for direct (peer-to-peer) shipping of data. Additional research is required to adapt our tools to support this scenario. This includes the development of deployment and management tools as well as a mapping of a process definition language that supports explicit handling of data (as it is typically used in scientific workflows), to EWFNs.

Support for (partial) process deployment in the "Cloud" We foresee extensions to our engine that provide the ability to move parts of the orchestration logic to the "cloud", e.g. for incremental outsourcing of parts of the process logic to servers hosted by cloud computing providers. A possible use case in this scenario is a step-by-step transition from on-premise hosting to cloud hosting of a single process model, by creating multiple partitions of the process and migrating each partition at a time. Even inside the cloud, flexible re-organization of computing resources (e.g. adding compute units in cases of high loads) that enact the process is possible using our execution model. To support this scenario, further research is required to solve problems like data ownership, dynamic

[1] `http://www.simtech.uni-stuttgart.de/?lang=en`

addition of nodes, adaptation or re-creation of EWFN partitions (possibly at runtime), management and monitoring tools, etc.

Support for additional workflow languages In our work, we have translated BPEL 2.0 to EWFNs in order to be able to execute BPEL processes in a decentralized manner. It would be interesting to translate other executable process definition languages, specifically the upcoming executable (i.e. Level 3) BPMN 2.0 [Sil09]. Interestingly, the semantics of BPMN 2.0 are defined based on token passing, and thus are based on the same principle as EWFNs.

Directly validate EWFNs We have defined a mapping from $\text{EWFN}_{\text{Linda}}$ to PT-nets. Using this mapping, standard Petri net analysis techniques can be applied to models expressed in $\text{EWFN}_{\text{Linda}}$. However, these techniques could be adapted to operate directly on the EWFN level in order to avoid the transformation step. Furthermore, it might be worth investigating how the additional information on the EWFN level, such as templates and tuplespace operations, help to improve accuracy of the validation algorithms and open the way for new properties to be analyzed. A first step into this direction was the definition of *conflict-free transitions*, *reducible distributed* `sync` *operations* and *conflict-free EWFNs* in Chapter 6.

Support for ad-hoc changes Each EWFN transition is only dependent on its direct pre- and post-set of places. A local change in the process model does thus not affect any transition that is not directly connected to the places affected by the change. This observation might be a good starting point for research of how support for ad-hoc changes can be integrated into the EWFN model, and how this affects the high-level process execution language used to create the EWFNs. Since EWFNs have formally defined semantics, change validation algorithms can be developed on their basis.

BIBLIOGRAPHY

[A+79] C. Alexander et al. *The Timeless Way of Building*. Oxford University Press, 1979.

[A+07] A. Arkin et al. WS-BPEL: Web Services Business Process Execution Language Version 2.0. OASIS Standard, `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, 2007.

[ÁBE05] P. Álvarez, J.A. Bañares, and J. Ezpeleta. Approaching Web Service Coordination and Composition by Means of Petri Nets. The Case of the Nets-within-Nets Paradigm. In *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[ACGV02] L. Aversano, A. Cimitile, P. Gallucci, and M.L. Villani. FlowManager: A Workflow Management System Based on Petri Nets. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC)*, 2002.

[AMG+95] G. Alonso, C. Mohan, R. Günthör, D. Agrawal, A. El Abbadi, and M. Kamath. Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management. In *Proceed-*

ings of the IFIP WG 8.1 Conference on Information Systems for Decentralized Organizations, 1995.

[AR01] E. Astesiano and G. Reggio. UML-SPACES: A UML Profile for Distributed Systems Coordinated via Tuple Spaces. In *Autonomous Decentralized Systems*, 2001.

[Arb04] F. Arbab. Reo: A Channel-based Coordination Model for Component Composition. In *Mathematical Structures in Computer Science*. Cambridge University Press, 2004.

[AS91] B. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In *Research Directions in High-Level Parallel Programming Languages*. Springer-Verlag, 1991.

[AvdADtH05] L. Aldred, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. On the Notion of Coupling in Communication Middleware. In *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[B+02] T. Bellwood et al. UDDI Version 3.0. Technical Committee Draft, OASIS, `http://uddi.org/pubs/uddi_v3.htm`, 2002.

[B+03] J. Billington et al. The Petri Net Markup Language: Concepts, Technology, and Tools. In *Proceedings of the International Conference on Application and Theory of Petri Nets and other Models of Concurrency (ICATPN)*. Springer-Verlag, 2003.

[B+04] D. Box et al. Web Services Addressing (WS-Addressing). W3C Member Submission, `http://www.w3.org/Submission/ws-addressing/`, 2004.

[B+06] K. Ballinger et al. WS-I Basic Profile Version 1.1. WS-I Organisation, `http://www.ws-i.org/Profiles/BasicProfile-1.1.html`, 2006.

[Bae05] J.C.M. Baeten. A Brief History of Process Algebra. In *Theoretical Computer Science*. North-Holland Publishing Company, 2005.

[BCF04] N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C#. In *ACM TOPLAS*, 2004.

[BCGZ01] N. Busi, P. Ciancarini, R. Gorrieri, and G. Zavattaro. Coordination Models: A Guided Tour. In *Coordination of Internet Agents: Models, Technologies, and Applications*. Springer-Verlag, 2001.

[BD97] T. Bauer and P. Dadam. A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration. In *Proceedings of the 2nd Conference on Cooperative Information Systems (Coopis)*, 1997.

[BD00] T. Bauer and P. Dadam. Efficient Distributed Workflow Management Based on Variable Server Assignments. In *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[BdCdM92] L. Bemardinello, F. de Cindio, and U.S. di Milano. A Survey of Basic Net Models and Modular Net Classes. In *Advances in Petri Nets*. Springer-Verlag, 1992.

[BGZ97] N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing Equivalence of Linda Coordination Primitives. In *Electronic Notes in Theoretical Computer Science*. Elsevier, 1997.

[BGZ98] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. In *Theoretical Computer Science*. Elsevier, 1998.

[BHR95] B. Blakely, H. Harris, and L. Rhys. *Messaging and Queueing Using the MQI: Concepts & Analysis, Design & Development*. McGraw-Hill, 1995.

[BMM06] L. Baresi, A. Maurino, and S. Modafferi. Towards Distributed BPEL Orchestrations. In *Electronic Communications of the EASST*, 2006.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*. Wiley & Sons, 1996.

[BN97] P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997.

[BP06] A. Brogi and R. Popescu. From BPEL Processes to YAWL Workflows. In *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[BR06] W. Brauer and W. Reisig. Carl Adam Petri und die "Petrinetze". In *Informatik-Spektrum*. Springer-Verlag, 2006.

[BRV04] B. Berthomieu, P.O. Ribet, and F. Vernadat. The Tool TINA - Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. In *International Journal of Production Research*, 2004.

[BV04] P. Buhler and J.M. Vidal. Enacting BPEL4WS Specified Workflows with Multiagent Systems. In *Proceedings of the Workshop on Web Services and Agent-Based Engineering*, 2004.

[BVV03] P. Buhler, J.M. Vidal, and H. Verhagen. Adaptive Workflow = Web Services + Agents. In *Proceedings of the International Conference on Web Services (ICWS)*. CSREA Press, 2003.

[BWA94] B. Butcher, A. Wood, and M. Atkins. Global Synchronization in Linda. In *Concurrency: Practice and Experience*, 1994.

[BWR08] A. Barker, C.D. Walton, and D. Robertson. Choreographing Web Services. In *Transactions on Services Computing*. ScholarOne, 2008.

[C+99] J. Clark et al. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, `http://www.w3.org/TR/xslt`, 1999.

[C+06] D. Churches et al. Programming Scientific and Distributed Workflow with Triana Services. In *Concurrency and Computation: Practice and Experience*. Wiley & Sons, 2006.

[C+07a] R. Chinnici et al. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C Recommendation, `http://www.w3.org/TR/wsdl20/`, 2007.

[C+07b] R. Chinnici et al. Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts. W3C Recommendation, `http://www.w3.org/TR/wsdl20-adjuncts/`, 2007.

[CCMN04] G.B. Chafle, S. Chandra, V. Mann, and M.G. Nanda. Decentralized Orchestration of Composite Web Services. In *Proceedings of the 13th International World Wide Web Conference (WWW)*. ACM Press, 2004.

[CDRV06] M. Cabano, E. Denti, A. Ricci, and M. Viroli. Designing a BPEL Orchestration Engine Based on ReSpecT Tuple Centres. In *Electronic Notes in Theoretical Computer Science*, 2006.

[CFN07] F.D. Cicirelli, A. Furfaro, and L. Nigro. A Service-based Control Engine for the Distributed Execution of Petri Nets and its Application to Workflow Management Systems. In *Proceedings of the 19th European Modelling and Simulation Symposium (EMSS)*, 2007.

[CFN08] F.D. Cicirelli, A. Furfaro, and L. Nigro. A Service-Based Enactment Engine for Dynamically Reconfigurable Workflows. In *Proceedings of the European Simulation and Modelling Conference (ESM)*, 2008.

[Cha04] D. Chappell. *Enterprise Service Bus*. O'Reilly, 2004.

[CKNW06] F. Curbera, R. Khalaf, W.A. Nagy, and S. Weerawarana. Implementing BPEL4WS: The Architecture of a BPEL4WS Implemen-

tation. In *Concurrency and Computation: Practice and Experience*. Wiley & Sons, 2006.

[CKS09] S. Craß, E. Kühn, and G. Salzer. Algebraic Foundation of a Data Model for an Extensible Space-Based Collaboration Protocol. In *Proceedings of the International Database Engineering & Applications Symposium (IDEAS)*. ACM Press, 2009.

[CLZ00] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A Programmable Coordination Architecture for Mobile Agents. In *IEEE Internet Computing*, 2000.

[Coa99] Workflow Management Coalition. Workflow Management Coalition Terminology & Glossary. WfMC Technical report, Document Number WFMC-TC-1011, Issue 3.0, `http://www.wfmc.org/index.php?option=com_docman&task=cat_view&gid=35&Itemid=74`, 1999.

[CR97] P. Ciancarini and D. Rossi. Jada-Coordination and Communication for Java Agents. In *Mobile Object Systems: Towards the Programmable Internet*, 1997.

[D+05a] E. Deelman et al. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. In *Scientific Programming*, 2005.

[D+05b] M. Dumas et al. Event-based Coordination of Process-Oriented Composite Applications. In *Proceedings of the International Conference on Business Process Management (BPM)*. Springer-Verlag, 2005.

[DE95] J. Desel and J. Esparza. *Free Choice Petri Nets*. Cambridge University Press, 1995.

[Dij67] E.W. Dijkstra. The Structure of the "THE"-Multiprogramming System. In *Proceedings of the first ACM Symposium on Operating System Principles*. ACM Press, 1967.

[DLO+08] G. Decker, A. Lueders, H. Overdick, K. Schlichting, and M. Weske. RESTful Petri Net Execution. In *Proceedings of the 5th International Workshop on Web Services and Formal Methods (WS-FM)*, 2008.

[DNO98] E. Denti, A. Natali, and A. Omicini. On the Expressive Power of a Language for Programming Coordination Media. In *Proceedings of the Annual ACM Symposium on Applied Computing (ACM SAC)*. ACM Press, 1998.

[DvdAtH05] M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley & Sons, 2005.

[FÁBE06] J. Fabra, P. Álvarez, J. Bañares, and J. Ezpeleta. RLinda: A Petri Net Based Implementation of the Linda Coordination Paradigm for Web Services Interactions. In *E-Commerce and Web Technologies*, 2006.

[FÁE07] J. Fabra, P. Álvarez, and J. Ezpeleta. DRLinda: A Distributed Message Broker for Collaborative Interactions Among Business Processes. In *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[FAH99] E. Freeman, K. Arnold, and S. Hupfer. *Javaspaces: Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[FG96] C. Fournet and G. Gonthier. The Reflexive CHAM and the Join-Calculus. In *Proceedings of 23rd ACM SIGPLAN-SIGACT*. ACM Press, 1996.

[FK97] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. In *International Journal of High Performance Computing Applications*, 1997.

[FK04] I. Foster and C. Kesselman. *The Grid: Blueprint for a new Computing Infrastructure*. Morgan Kaufmann, 2004.

[FKLT07] D. Fensel, E. Kühn, F. Leymann, and R. Tolksdorf. Queues Are Spaces - Yet Still Both Are Not The Same? Technical report, EU FP6 Project TripCom, IST-4-027324-STP, `http://tripcom.org/docs/sbc2007.pdf`, 2007.

[FKS+07] D. Fensel, R. Krummenacher, O. Shafiq, E. Kuehn, J. Riemer, Y. Ding, and B. Draxler. TSC - Triple Space Computing. In *Special Issue on ICT Research in Austria, Journal of Electronics & Information Technology*, 2007.

[FLN+03] M. Fontoura, T. Lehman, D. Nelson, T. Truong, and Y. Xiong. TSpaces Services Suite: Automating the Development and Management of Web Services. In *Proceedings of the 12th International World Wide Web Conference (WWW)*. Citeseer, 2003.

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[Fre02] J. Frey. Condor DAGMan: Handling Inter-Job Dependencies. Technical report, University of Wisconsin, Department of Computer Science, `http://www.cs.wisc.edu/condor/dagman`, 2002.

[FTL+02] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Cluster Computing*. Springer-Verlag, 2002.

[GC92] D. Gelernter and N. Carriero. Coordination Languages and their Significance. In *Communications of the ACM*. ACM Press, 1992.

[Gel85] D. Gelernter. Generative Communication in Linda. In *ACM Transactions on Programming Languages and Systems*. ACM Press, 1985.

[Gen86] H.J. Genrich. Predicate/Transition Nets. In *Advances in Petri Nets*. Springer-Verlag, 1986.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GLO98] V. Guth, K. Lenz, and A. Oberweis. Distributed Workflow Execution Based on Fragmentation of Petri Nets. In *Proceedings of 15th IFIP World Computer Congress Telecooperation*, 1998.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gra95] J. Gray. Queues are Databases. Technical report, MSR-TR-95-56, Microsoft Research, Microsoft Corporation, Redmond, USA, 1995.

[Gra97] B. Grahlmann. The PEP Tool. In *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[GRCB05a] L. Guo, D. Robertson, and Y. Chen-Burger. A Novel Approach for Enacting the Distributed Business Workflows Using BPEL4WS on the Multi-Agent Platform. In *Proceedings of the IEEE International Conference on e-Business Engineering*, 2005.

[GRCB05b] L. Guo, D. Robertson, and Y. Chen-Burger. Enacting the Distributed Business Workflows using BPEL4WS on the Multi-Agent Platform. In *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. In *Communications of the ACM*. ACM Press, 1978.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Hol97] G.J. Holzmann. The Model Checker SPIN. In *IEEE Transactions on Software Engineering*, 1997.

[HSS05] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *Proceedings of the International Conference on Business Process Management (BPM)*. Springer-Verlag, 2005.

[HWB03] G. Hohpe, B. Woolf, and K. Brown. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[IK02] G.S. Itzstein and D. Kearney. Applications of Join Java. In *Proceedings of the 7th Asia-Pacific Conference on Computer Systems Architecture*, 2002.

[Jen92] K. Jensen. Coloured Petri Nets, Vol. 1: Basic Concepts. In *EATCS Monographs on Theoretical Computer Science*, 1992.

[JKMUN02] T. Jacob, O. Kummer, D. Moldt, and U. Ultes-Nitsche. Implementation of Workflow Systems Using Reference Nets – Security and Operability Aspects. In *Proceedings of the 4th International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools*, 2002.

[JLN08] H. Jorgensen, O. Lawrence, and A. Neus. Making Change Work. IBM Global Business Services, `http://www-935.ibm.com/services/us/gbs/bus/html/gbs-making-change-work.html`, 2008.

[JPPMMJ08] R. Jiménez-Peris, M. Patiño-Martínez, and E. Martel-Jordán. Decentralized Web Service Orchestration: A Reflective Approach. In *Proceedings of the Annual ACM Symposium on Applied Computing (ACM SAC)*. ACM Press, 2008.

[Küh94] E. Kühn. Fault-Tolerance for Communicating Multidatabase Transactions. In *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS)*, 1994.

[Küh03] E. Kühn. The Zero-Delay Data Warehouse: Mobilizing Heterogeneous Databases. In *Proceedings of the 29th Conference on Very Large Data Bases (VLDB)*, 2003.

[Kha07] R. Khalaf. Note on Syntactic Details of Split BPEL-D Business Processes. Technical report, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, Germany, `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2007-02&engl=1`, 2007.

[Kha08] R. Khalaf. *Supporting Business Process Fragmentation While Maintaining Operational Semantics - A BPEL Perspective*. PhD thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, Germany, `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIS-2008-04&mod=0&engl=1`, 2008.

[KKL08] O. Kopp, R. Khalaf, and F. Leymann. Deriving Explicit Data Links in WS-BPEL Processes. In *Proceedings of the IEEE International Conference on Services Computing (SCC)*, 2008.

[KKS+06] D. Karastoyanova, R. Khalaf, R. Schroth, M. Paluszek, and F. Leymann. BPEL Event Model. Technical report, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, Germany, `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2006-10&engl=1`, 2006.

[KL06] R. Khalaf and F. Leymann. Role-based Decomposition of Business Processes using BPEL. In *Proceedings of the International Conference of Web Services (ICWS)*, 2006.

[KLF04] D. Khushraj, O. Lassila, and T. Finin. sTuples: Semantic Tuple Spaces. In *Proceedings of the 1st Annual International Conference*

on Mobile and Ubiquitous Systems: Networking and Services, 2004.

[KMKS09] E. Kühn, R. Mordinyi, L. Keszthelyi, and C. Schreiber. Introducing the Concept of Customizable Structured Spaces for Agent Coordination in the Production Automation Domain. In *Proceedings of 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. Springer-Verlag, 2009.

[KML08] O. Kopp, R. Mietzner, and F. Leymann. Abstract Syntax of WS-BPEL 2.0. Technical report, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, Germany, `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2008-06&engl=1`, 2008.

[KMS08] E. Kühn, R. Mordinyi, and C. Schreiber. An Extensible Space-Based Coordination Approach for Modeling Complex Patterns in Large Systems. In *Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Springer-Verlag, 2008.

[KMWL08] O. Kopp, D. Martin, D. Wutke, and F. Leymann. On the Choice Between Graph-Based and Block-Structured Business Process Modeling Languages. In *Modellierung betrieblicher Informationssysteme (MobIS)*, Lecture Notes in Informatics. Gesellschaft für Informatik e.V. (GI), 2008.

[KRJ05] E. Kühn, J. Riemer, and G. Joskowicz. XVSM (eXtensible Virtual Shared Memory) - Architecture and Application. Technical report, E185/1, University of Vienna, SBC-Group, Austria, 2005.

[KS09] E. Kühn and V. Sesum-Cavic. A Space-Based Generic Pattern for Self-Initiative Load Balancing Agents. In *Proceedings of the 10th International Workshop on Engineering Societies in the Agents World*. Springer-Verlag, 2009.

[Kum02] O. Kummer. *Referenznetze*. Logos-Verlag, 2002.

[KW01] E. Kindler and M. Weber. The Petri Net Kernel: An Infrastructure for Building Petri Net Tools. In *International Journal on Software Tools for Technology Transfer (STTT)*. Springer-Verlag, 2001.

[KWD+04] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Kohler, D. Moldt, H. Rolke, and R. Valk. An Extensible Editor and Simulation Engine for Petri Nets: Renew. In *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[L+06] B. Ludascher et al. Scientific Workflow Management and the Kepler System. In *Concurrency and Computation: Practice and Experience*. Wiley & Sons, 2006.

[LA07] A. Lazovik and F. Arbab. Using Reo for Service Coordination. In *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[Ley03] F. Leymann. Web Services: Distributed Applications without Limits. In *Datenbanksysteme in Büro, Technik und Wissenschaft - Business, Technology and Web (BTW)*, 2003.

[Ley05] F. Leymann. The (Service) Bus: Services Penetrate Everyday Life. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*, 2005.

[Ley06] F. Leymann. Space-based Computing and Semantics: A Web Service Purist's Point-Of-View. Technical report, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, Germany, `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2006-05&engl=1`, 2006.

[LH+04] A. Le Hors et al. Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation, `http://www.w3.org/TR/DOM-Level-3-Core/`, 2004.

[LMSW06] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing Interacting BPEL Processes. In *Proceedings of the International Conference on Business Process Management (BPM)*. Springer-Verlag, 2006.

[LMW99] T.J. Lehman, S.W. McLaughry, and P. Wyckoff. TSpaces: The Next Wave. In *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*, 1999.

[Loh08] N. Lohmann. A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In *International Workshop on Web Services and Formal Methods (WS-FM)*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

[LR97] F. Leymann and D. Roller. Workflow-Based Applications. In *IBM Systems Journal: Application Development*, 1997.

[LR98] F. Leymnn and D. Roller. Building a Robust Workflow Management System with Persistent Queues and Stored Procedures. In *Proceedings of the 14th International Conference on Data Engineering*, 1998.

[LR00] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000.

[LSW97] P. Langner, C. Schneider, and J. Wehler. Prozessmodellierung mit ereignisgesteuerten Prozessketten (EPKs) und Petri-Netzen. In *Wirtschaftsinformatik*, 1997.

[LVD09] N. Lohmann, E. Verbeek, and R. Dijkman. Petri Net Transformations for Business Processes - A Survey. In *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[MdFK+08] D. Martin, D. de Francisco, R. Krummenacher, H. Moritsch, and D. Wutke. An Architecture for a QoS-Aware Application Integration Middleware. In *Proceedings of the 11th International*

Conference on Business Information Systems (BIS). Springer-Verlag, 2008.

[MHC00] R. Monson-Haefel and D. Chappell. *Java Message Service*. O'Reilly, 2000.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1982.

[Mil99] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[MRS05] P. Massuthe, W. Reisig, and K. Schmidt. *An Operating Guideline Approach to the SOA*. Annals of Mathematics, Computing & Teleinformatics, 2005.

[MvDvdA07] J. Mendling, B.F. van Dongen, and W.M.P. van der Aalst. Getting Rid of the OR-Join in Business Process Models. In *Proceedings of the IEEE International EDOC Conference (EDOC)*, 2007.

[MWL08a] D. Martin, D. Wutke, and F. Leymann. A Novel Approach to Decentralized Workflow Enactment. In *Proceedings of the IEEE International EDOC Conference (EDOC)*, 2008.

[MWL08b] D. Martin, D. Wutke, and F. Leymann. EWFN - A Petri Net Dialect for Tuplespace-based Workflow Enactment. In *Proceedings of the 15th German Workshop on Algorithms and Tools for Petri Nets (AWPN)*, CEUR Workshop Proceedings. CEUR-WS.org, 2008.

[MWL08c] D. Martin, D. Wutke, and F. Leymann. Synchronizing Control Flow in a Tuplespace-Based, Distributed Workflow Management System. In *Proceedings of the 10th International Conference on Electronic Commerce (ICEC)*. ACM Press, 2008.

[MWL08d] D. Martin, D. Wutke, and F. Leymann. Using Tuplespaces to Enact Petri Net-Based Workflow Definitions. In *Proceedings of*

the 10th International Conference on Information Integration and Web-based Applications & Services (iiWAS), 2008.

[MWL09] D. Martin, D. Wutke, and F. Leymann. Tuplespace Middleware for Petri Net-Based Workflow Execution. In *International Journal on Web and Grid Services (IJWGS)*. Inderscience Publishers, 2009.

[MWSL07] D. Martin, D. Wutke, T. Scheibler, and F. Leymann. An EAI Pattern-Based Comparison of Spaces and Messaging. In *Proceedings of the IEEE International EDOC Conference (EDOC)*, 2007.

[NCS04] M.G. Nanda, S. Chandra, and V. Sarkar. Decentralizing Execution of Composite Web Services. In *ACM SIGPLAN Notices*. ACM Press, 2004.

[OH98] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. Wiley & Sons, 1998.

[OVvdA+05a] C. Ouyang, E. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, and A.H.M. ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. In *Science of Computer Programming*, 2005.

[OVvdA+05b] C. Ouyang, E. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, and A.H.M. ter Hofstede. WofBPEL: A Tool for Automated Analysis of BPEL Processes. In *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[OZ98] A. Omicini and F. Zambonelli. TuCSoN: A Coordination Model for Mobile Information Agents. In *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, 1998.

[OZKT01] A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf. *Coordination of Internet Agents: Models, Technologies, and Applications*. Springer-Verlag, 2001.

[PA98] G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. In *The Engineering of Large Systems*. Academic Press, 1998.

[Pap08] M. Papazoglou. *Web Services: Principles and Technology*. Addison-Wesley, 2008.

[Pet62] C.A. Petri. Kommunikation mit Automaten (Communication with Automata). In *Schriften des Rheinisch-Westfälischen Instituts für Instrumentelle Mathematik, Bonn, Germany*, 1962.

[Pop08] P. Popova. Transformation von BPEL Prozessen in ein verteiltes Ausführungsmodell. Master's thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, Germany, `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2802&engl=1`, 2008.

[RA01] G. Reggio and E. Astesiano. An Extension of UML for Modelling the non Purely-Reactive Behaviour of Active Objects. In *Semi-Formal and Formal Specification Techniques for Software Systems, Dagstuhl Seminar*. Citeseer, 2001.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Rei85] W. Reisig. *Petri Nets: An Introduction*. Springer-Verlag, 1985.

[RMRD+06] J. Riemer, F. Martin-Recuerda, Y. Ding, M. Murth, B. Sapkota, R. Krummenacher, O. Shafiq, D. Fensel, and E. Kühn. Triple Space Computing: Adding Semantics to Space-Based Computing. In *Proceedings of the Asian Semantic Web Conference (ASWC)*, 2006.

[RtHvdAM06] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical report, BPM Center Report BPM-06-22,

BPMCenter.org, http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-22.pdf, 2006.

[RW96] A. Rowstron and A. Wood. Solving the Linda Multiple rd Problem. In *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[RW97] A.I.T. Rowstron and A.M. Wood. BONITA: A Set of Tuple Space Primitives for Distributed Coordination. In *Proceedings of the 30th HICSS: Software Technology and Architecture*, 1997.

[Sch00] K. Schmidt. LoLA: A Low Level Analyser. In *Proceedings of 21st International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN)*. Springer-Verlag, 2000.

[Sch04] C. Schuler. *Verteiltes Peer-to-Peer-Prozessmanagement - Die Realisierung einer Hyperdatenbank*. PhD thesis, Eidgenössische Technische Hochschule Zürich, Switzerland, 2004.

[Sch07] A. Schwind. Space-Based Web Services: Konzepte und prototypische Implementierung mit Linda-Spaces. Master's thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, Germany, http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2692&engl=1, 2007.

[SGK+10] M. Sonntag, K. Görlach, D. Karastoyanova, F. Leymann, and M. Reiter. Process Space-Based Scientific Workflow Enactment. In *International Journal of Business Process Integration and Management (IJBPIM)*. Inderscience Publishers, 2010.

[Sil09] B. Silver. *BPMN Method and Style*. Cody-Cassidy Press, 2009.

[SKS98] A. Silberschatz, H.F. Korth, and S. Sudershan. *Database System Concepts*. McGraw-Hill, 1998.

[Sta05] C. Stahl. A Petri Net Semantics for BPEL. Technical report, Humboldt-Universität zu Berlin, Department of Computer Science, 2005.

[Ste08] T. Steinmetz. Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE. Master's thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, Germany, `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2729&engl=1`, 2008.

[Syr01] A. Syropoulos. Mathematics of Multisets. In *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[T+04] H. S. Thompson et al. XML Schema (XSD) Version 1.0. W3C Recommendation, `http://www.w3.org/XML/Schema`, 2004.

[TG01] R. Tolksdorf and D. Glaubitz. Coordinating Web-Based Systems with Documents in XMLSpaces. In *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[Tol02] R. Tolksdorf. Workspaces: A Web-Based Workflow Management System. In *IEEE Internet Computing*, 2002.

[Val98] R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In *Proceedings of the 19th International Conference on Application and Theory of Petri Nets*. Springer-Verlag, 1998.

[VBvdA01] H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing Workflow Processes using Woflan. In *The Computer Journal*. British Computer Society, 2001.

[vdA98] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. In *The Journal of Circuits, Systems and Computers*, 1998.

[vdAADtH04] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and Implementation of the YAWL system. In *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[vdAtH05] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. In *Information Systems*. Elsevier, 2005.

[vdAtHKB03] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. In *Distributed and Parallel Databases*. Springer-Verlag, 2003.

[VHvdA02] H.M.W. Verbeek, A. Hirnschall, and W.M.P. van der Aalst. XRL/Flower: Supporting Inter-Organizational Workflows Using XML/Petri-Net Technology. In *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[VSY98] W. Vogler, A. Semenov, and A. Yakovlev. Unfolding and Finite Prefix for Nets with Read Arcs. In *Proceedings of the 9th International Conference on Concurrency Theory*. Springer-Verlag, 1998.

[W+09] S.A. White et al. Business Process Modeling Notation (BPMN), Version 1.2. The Object Management Group (OMG), `http://www.omg.org/spec/BPMN/1.2/`, 2009.

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D.F. Ferguson. *Web Services Platform Architecture*. Prentice Hall PTR, 2005.

[Wes07] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag, 2007.

[WML08a] D. Wutke, D. Martin, and F. Leymann. Facilitating Complex Web Service Interactions through a Tuplespace Binding. In *Proceedings of the 8th IFIP WG 6.1 International Conference*

on Distributed Applications and Interoperable Systems (DAIS). Springer-Verlag, 2008.

[WML08b] D. Wutke, D. Martin, and F. Leymann. Model and Infrastructure for Decentralized Workflow Enactment. In *Proceedings of the Annual ACM Symposium on Applied Computing (ACM SAC)*. ACM Press, 2008.

[WML08c] D. Wutke, D. Martin, and F. Leymann. Triple Space Binding for Web Services. Technical report, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, Germany, `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2008-04&engl=1`, 2008.

[WML09a] D. Wutke, D. Martin, and F. Leymann. A Method for Partitioning BPEL Processes for Decentralized Execution. In *Proceedings of the 1st Central-European Workshop on Services and their Composition (ZEUS)*, CEUR Workshop Proceedings. CEUR-WS.org, 2009.

[WML09b] D. Wutke, D. Martin, and F. Leymann. Tuplespace-based Infrastructure for Decentralized Enactment of BPEL Processes. In *Proceedings of the 9th Internationale Tagung Wirtschaftsinformatik: Business Services, Konzepte, Technologien, Anwendungen (WI)*, 2009.

[Wu08] S. Wu. Verteilte Workflow-Engine: Ausführung verteilter Prozessmodelle auf Basis von Tuplespaces. Master's thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, Germany, `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2831&engl=1`, 2008.

[Wut06] D. Wutke. Erweiterung einer Workflow-Engine zur Unterstützung von parametrisierten Web Service Flows. Master's

thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, Germany, http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2401&engl=1, 2006.

[Wut10] D. Wutke. *Eine Infrastruktur für die dezentrale Ausführung von BPEL-Prozessen*. PhD thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering and Information Technology, Germany, 2010.

[Yil07] U. Yildiz. On Dead Path Elimination in Decentralized Process Executions. Technical report, INRIA, Université Henri Poincaré, Institut National Polytechnique de Lorraine, Nancy, France, 2007.

[Yu07a] W. Yu. Peer-to-Peer Execution of BPEL Processes. Technical report, Department of Computer Science, University of Tromso, Norway, 2007.

[Yu07b] W. Yu. Running BPEL Processes without Central Engines. In *Proceedings of Norsk Informatikkonferanse (NIK)*, 2007.

[Zlo77] M.M. Zloof. Query-by-Example: A Data Base Language. In *IBM Systems Journal*, 1977.

Hyperlinks were last followed on June 9, 2010.

List of Figures

List of Tables

List of Algorithms

List of Symbols and Primary Functions

Symbols and Primary Functions Used

Notation	Meaning
P	The set of places
T	The set of transitions
X	The set of templates
$F \subseteq (T \times P) \cup (P \times T \times R)$	The set of arcs, also called *flow relation*
R	The set of arcs that go from places to transitions. Initially, $R = \{\text{read}, \text{take}\}$. This is later extended to $R = \{\text{read}, \text{take}, \text{takeall}, \text{readall}, \text{update}, \text{sync}\}$
$A : (P \times T \times R) \rightarrow X$	A function that assigns a template $te \in X$ to each arc going from a place to a transition
$B : (T \times P) \rightarrow \mathcal{P}(\Phi)$	Assigns a set $c \in \mathcal{P}(\Phi)$ of *tuple types* to outgoing arcs of a transition
Φ	The set of tuple types, e.g. (int, string)

$I_0 : P \rightarrow \Sigma^{MS}$	The *initialization function* that assigns a multiset (indicated by ^{MS}) over Σ to places
$L_{\text{write}} : (T \times P) \rightarrow \Sigma$	The Linda `write` function (sometimes also called "tuple production rule") that determines the token to be written by each outgoing arc of a transition
TE	A *tuple element* TE is a tuple $\langle p, tu \rangle$, $p \in P$, $tu \in \Sigma$
$M \in \text{TE}^{MS}$	A *marking* M is a multiset over tuple elements
M_0	The *initial marking* M_0 is created when applying the initialization function to all places
$L_{\text{read}} : X \times \Sigma^{MS} \rightarrow \Sigma$	Linda `read` function
$\approx$	A binary relation over the sets Σ and X, specifying if a template matches a tuple: $\approx \subseteq \Sigma \times X$
te	A template, te $\in X$
tu	A tuple, tu $\in \Sigma$
PX	The set of *place elements*, defined as $\text{PX} = \{\langle p, x \rangle \mid p \in P, x = A(\langle p, t, r \rangle), t \in p\bullet\}$
TX	The set of *transition elements*, defined as $TX = \{\langle t, x \rangle \mid t \in T, x = B(\langle t, p \rangle), p \in t\bullet\}$
$\pi_i(t)$	Returns a projection to the ith element of a tuple t
$\mathcal{P}(S)$	Denotes the power set of a set S
Σ_{BPEL}	The set of tuples used for expressing BPEL as EWFN, $\Sigma_{\text{BPEL}} = \{CF, DATA \times \mathbb{N}, DATA \times \mathbb{N} \times \text{String}, \ldots, \epsilon\}$
`CF`	A control flow tuple, `CF` $= \langle$"CF", $\mathbb{B}, \mathbb{N}, \mathbb{N}, \langle \mathbb{N}, \mathbb{N}, \ldots \rangle \rangle$. Set $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ denotes either "positive" or negative control flow
`DP`	A control flow tuple with the second component set to `false` denoting negative control flow, i.e. a dead path

`DATA`	The generic part of a tuple representing process instance data. `DATA` $= \langle$"*DATA*", $\mathbb{N}, \mathbb{N}, \langle \mathbb{N}, \mathbb{N}, \ldots \rangle \rangle$. This part is prepended to each variable definition in tuple form
XMLstruct	Represents "the infinite set of XML-Schema types that can be defined using XML-Schema and are included in the process definition" [KML08]
F^{MS}	A multiset of tuples denoting the contents of a tuplespace
$tu \in \Sigma$	A tuple
$te \in X$	A template
$OP \in \{$"*read*", "*take*"$\}$	The set of possible operations to apply a template to the tuplespace
$K \subseteq X \times OP$	The set of parameter tuples given to a `sync` operation. $\langle te, op \rangle \in K$ is one parameter with op denoting the operation to use to apply template te
$H \subseteq X \times OP \times \mathbb{B}$	The set of active templates on the tuplespace, described in tuple form $\langle te, op, b \rangle$, $b \in \mathbb{B} = \{\texttt{true}, \texttt{false}\}$ indicates if the template $te \in X$ can be satisfied
$U \subseteq H$	The temporary set used to construct an element of set G
$G \subset \mathcal{P}(H)$	The set of sets of active templates, more specifically, the parameter sets of all tuple consuming operations currently active on the space. A `sync` operation with three templates for instance results in a set with three elements that is added to set G

$\texttt{check}_\texttt{t} : X \rightarrow \mathbb{B}$	Checks if template $te \in X$ can be satisfied with the current state of the space, i.e. $$\texttt{check}_\texttt{t}(te) = \begin{cases} \texttt{true}, & \text{if } \texttt{Int.Read}(te) \neq \epsilon \\ \texttt{false}, & \text{otherwise} \end{cases}$$
$\texttt{eval} : H \rightarrow \mathbb{N}$	An evaluation function that calculates a value used to select the blocking operation to wake-up if more than one may be woken up
$\texttt{max} : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$	Returns the maximum of two integers, i.e. $$\texttt{max}(n, m) = \begin{cases} n, & \text{if } n \geq m \\ m, & \text{otherwise} \end{cases}$$
$\texttt{count} : \Sigma \rightarrow \mathbb{N}$	Returns the highest number of equal elements in tuple $tu \in \Sigma$. Consider the tuple $tu = \langle 1, 1, 5, "a", "b" \rangle$ for instance, in this case $\texttt{count}(tu) = 2$
$L \subseteq X$	A set of templates with equal length (equal number of elements, i.e. $\exists n : \forall$ te $\in L : \lvert\text{te}\rvert = n$) for synchronized consumption. Note that the templates may contain join-variables
$\texttt{check}_{\texttt{t,jv}} : L \rightarrow \mathbb{B}$	The function that handles the matching for join-variables (cf. Algorithm 6.17)
$\circ$	The concatenation operator for tuples, for instance $\langle a, b \rangle \circ \langle 1, 2 \rangle = \langle a, b, 1, 2 \rangle$
$\mathbb{N}^*$	Denotes the set of natural numbers *without* 0
$\mathbb{N}_0$	Denotes the set of natural numbers *including* 0

Acronyms

BPEL	see WS-BPEL
BPM	Business Process Management
CHIG	Compensation Handler Instance Group
CP-net	Colored Petri Net
CPN	same as CP-net
DBMS	Database Management System
DCO	Default Compensation Order
DPE	Dead-Path-Elimination
EAI	Enterprise Application Integration
ebXML	e-business XML
EI	Enterprise Integration
EJB	Enterprise Java Bean
ESB	Enterprise Service Bus
EWFN	Executable Workflow Network
FIPA	Foundation for Intelligent Physical Agents
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocoll
i/o	Input / Output
ID	Identification
IDE	Integrated Development Environment

IMA	Inbound Message Activity
JBI	Java Business Integration
JDBC	Java Database Connectivity
JEE	Java Enterprise Edition
JMS	Java Message Service
JTA	Java Transaction API
KPI	Key Performance Indicator
MOM	Message-Oriented Middleware
MQI	Message Queuing Interface
OASIS	Organization for the Advancement of Structured Information Standards
OMG	Object Management Group
OS	Operating System
PT-net	Place-Transition Net
QoS	Quality of Service
RDBMS	Relational Database Management System
SaaS	Software as a Service
SBC	Space-based computing
SLA	Service Level Agreement
SOA	Service-oriented Architecture
SOAP	Ealier: Simple Object Access Protocoll, now stands for itself
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
URI	Universal Resource Identifier
W3C	World Wide Web Consortium
WfMS	Workflow Management System
WS	Web Service
WS-AT	Web Service Atomic Transaction
WS-BA	Web Service Business Activity
WS-BPEL	Web Services Business Process Execution Language
WS-CDL	Web Service Choreography Description Language
WS-I	Web Service Interoperability
WS-RF	Web Service Resource Framework

WSDL	Web Service Description Language
WSFL	Web Service Flow Language
WSIF	Web Services Invocation Framework
XLang	eXtensible Language
XML	Extensible Markup Language
XPDL	XML Process Definition Language

Index