2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering

Workflow Semantics of Peer and Service Behaviour

Antonio Brogi and Razvan Popescu*
Computer Science Department, University of Pisa, Italy

Abstract

In this paper we introduce SMoL, a simplified BPEL-like language for specifying peer and service behaviour in P2P systems. We define a transformational semantics of SMoL in terms of YAWL workflows, which enables the simulation (e.g., testing possible execution scenarios) and analysis (e.g., verifying reachability or lock freedom) of the behaviour of P2P peers and services.

1 Introduction

The goal of the Secure Middleware for Embedded Peer-To-Peer Systems (SMEPP) Project (`www.smepp.org`, [1]) is to develop a middleware that will have to be secure, generic and highly customisable, allowing for its adaptation to different devices (from PDAs and smart phones to embedded sensor actuator systems) and to different domains (from critical systems to consumer entertainment or communication). Furthermore, SMEPP aims to provide a high-level, service-oriented model to program the interaction among peers, thus hiding low-level details that concern the supporting infrastructure.

The key features of the model are the notion of *group of peers*, the notion of *service* offered by peers (or by groups), and the concern of *security*. In short, the model defines a set of security-aware *primitives* for peer management (e.g., create peers), for group management (e.g., create, join, or leave groups), for service management (e.g., publish and unpublish services), and for message and event handling (viz., send or receive messages, or subscribe, unsubscribe, raise, and receive events), to be implemented by one or more APIs. Such primitives are the basic bricks for constructing the code of P2P entities[1].

The SMEPP model is equipped with a high-level language (SMoL — SMEPP Modelling Language) for specifying how to orchestrate SMEPP primitives into peer or service code. The availability of a high-level specification language notably simplifies the time-consuming and error-prone task of specifying the interactions of a complex P2P system. Most importantly, the definition of a formal semantics for such a language in term of workflows (the subject of this paper) enables the simulation and the analysis of the behaviour of peers and services, thus featuring the possibility of developing not only secure, but also *a priori* verified SMEPP specifications. Furthemore, the availability of automatic translators (e.g., the prototype SMoL2Java compiler) greatly simplifies the generation of executable code, which can be further completed to express data-related details in peer/service behaviour.

This paper briefly describes the SMEPP primitives and modelling language (Section 2), and it then focuses on the translation of SMoL programs into YAWL workflows (Section 3). Section 4 briefly describes the translation of a simple SMoL example. Section 5 presents some concluding remarks.

2 SMEPP Primitives and Modelling Language

The analysis of current state-of-the-art models in P2P systems (e.g., [2, 3, 6, 7, 8, 10]) revealed the fact that existing frameworks for the development of P2P applications generally: (1) do not provide a simple, high-level service (interaction) model that presents a suitable level of abstraction to ease the development of P2P applications, or (2) do not model all the concepts mentioned in Section 1 (such as group management, or message and event handling), or (3) do not provide a (formal) abstract language that can be used for simulating and verifying the behaviour of peers and services, and their interactions, as well as for application prototyping.

In order to tackle such limitations we defined a SMEPP service model that features a set of abstract primitives, which can be used to develop P2P application specifications in a simple, high-level manner. We aim at deploying such primitives as different (language dependent) APIs, which will allow the deployment of SMEPP specifications as real (executable) applications.

*Work partly supported by the SMEPP project (EU-FP6-IST 0333563).

[1] We shall use the term "entity" to refer to peers or services.

978-0-7695-3249-3/08 $25.00 © 2008 IEEE
DOI 10.1109/TASE.2008.28

2.1 SMEPP Primitives

```
// Peer Management
pId newPeer(creds)
pId getPeerId(id?)
pId[] getPeers(gId)

// Group Management
gId createGroup(grDescr)
gId[] getGroups(grDescr?)
grDescr getGroupDescription(gId)
void joinGroup(gId, creds)
void leaveGroup(gId)
gId[] getIncludingGroups()
gId getPublishingGroup(id?)

// Service Management
<gSId, pSId> publish(gId, contract)
void unpublish(pSId)
<gId, gSId, pSId>[]
      getServices(gId?, pId?, contract?, maxRes?, creds)
contract getServiceContract(id)
sessId startSession(sId)

// Message Handling
out? invoke(eId, opName, in?)
<cId, in?> receiveMessage(gId?, opName)
void reply(cId, opName, out?, fName?)

// Event Handling
void subscribe(evName?, gId?)
void unsubscribe(evName?, gId?)
void event(gId?, evName, in?)
<cId, in?> receiveEvent(gId?, evName)
```

Figure 1. SMEPP Primitives.

The SMEPP primitives are given in Figure 1[2]. Basically, the SMEPP model identifies peers, groups, and services through unique identifiers. The SMEPP primitives offer support for: (1) *peer management*: `newPeer`, `getPeerId`, and `getPeers`, (2) *group management*: `createGroup`, `getGroups`, `getGroupDescription`, `joinGroup`, `leaveGroup`, `getIncludingGroups`, and `getPublishingGroup`, (3) *service management*: `publish`, `unpublish`, `getServices`, `getServiceContract`, and `startSession`, (4) *message handling*: `invoke`, `receiveMessage`, and `reply`, and for (5) *event handling*: `subscribe`, `unsubscribe`, `event`, and `receiveEvent`.

2.2 SMoL: SMEPP Modelling Language

SMoL defines the behaviour of the SMEPP entities as compositions of *basic commands* into *structured* ones. SMoL is inspired by version 2.0 of BPEL [12], which recently became the OASIS standard for describing Web service compositions.

Since the BPEL semantics [4] is quite complex (e.g., due to synchronisation links and dead-path-elimination), the analysis of (interactions of) BPEL processes is both troublesome and very time consuming. Furthermore, the SMEPP requirements do not request several BPEL constructs (concepts). Consequently, we designed SMoL from BPEL basically by removing the following BPEL concepts: compensations, synchronisation links (and hence dead-path-elimination), the forEach construct, serializable scopes, partner links, message properties, and correlation sets.

Furthermore, by employing the SMEPP primitives as basic SMoL commands, we allow the SMoL programs to manage: peers (e.g., peer creation), groups (e.g., group creation and discovery), services (e.g., service publication and discovery, state-less and state-full services), messages (viz., one-way and request-response operation invocations), and events (non-blocking, asynchronous communication; event generators and subscribers).

In the following, we describe the basic and the structured SMoL commands.

2.2.1 Basic Commands.

```
primitive
void empty()
void wait(for?, until?, repeatEvery?)
void throw(faultName, faultVariable?)
faultVariable? catch(faultName)
<faultName, faultVariable?> catchAll()
void exit()
```

Figure 2. SMoL Basic Commands.

Figure 2 illustrates SMoL basic commands. A basic command is eithera call to any of the `primitive`s shown in Figure 1, or a call to `empty`, `wait`, `throw`, `catch`, `catchAll`, or `exit`.

As previously mentioned, SMEPP primitives are basic commands. `Empty` is equivalent to a no-op. `Wait` delays the execution of the caller either for a time interval (viz., "for"), or until reaching a certain moment in time (viz., "until"). The repeatEvery parameter serves to repeat the delay (see informationHandler in Subsection 2.2.2). `Throw` raises an (explicit) fault inside the caller's program; faultVariable identifies the data associated to the fault. `Catch/CatchAll` serve to catch and process faults raised inside the caller's program (see the faultHandler in Subsection 2.2.2). `Exit` terminates the execution of the peer or service code.

[2]The question mark denotes optional parameters, square brackets represent arrays, and angle brackets composite data structures.

2.2.2 Structured Commands.

Figure 3 illustrates SMoL structured commands[3]. `Sequence` provides basic sequential control-flow, while `flow` provides concurrency in the control-flow of the modelled program. `While` and `repeatUntil` provide looping mechanisms. `If` provides conditional control-flow (deterministic choice). `Assign` defines one or more `copy` commands, which copy the value of the source `from` into the target `to` variable. `From` can specify a variable, an expression, or a literal (inline text to be assigned to the target variable, such as a number). The `opaque` keyword serves to hide the source of the assignment. `Pick` defines a non-deterministic global choice in the control-flow of the modelled program. In short, the execution of the pick resumes to executing a message or event branch upon the reception of a corresponding message or event. Furthermore, the expiration of a timer triggers the corresponding wait (viz., alarm) branch. `InformationHandler` is somewhat similar to a pick. However, the informationHandler has an associated command that dictates its lifetime. As long as the associated command executes, the informationHandler can execute message, event, or alarm branches (one or more times). The `repeatEvery` keyword sets the time interval after which the alarm goes off periodically. `FaultHandler` also encloses an associated command. A fault raised by the execution of the associated command triggers the execution of the first matching (in lexical order) `catch` or `catchAll` branch. `Catch` basically matches the name of the fault, while `catchAll` matches every fault. In case no such match exists, the fault is propagated at the enclosing `faultHandler`. In [5] we proposed a first semantics for (an abstract version of) SMoL. Such semantics formally establishes whether a set of SMEPP processes (viz., peer or service codes) can be executed together without locking, and it provides a solid ground to develop tools for the analysis and verification of SMEPP specifications.

3 Translating SMoL Programs into YAWL workflows

3.1 A Brief Introduction to YAWL

The Yet Another Workflow Language (YAWL) is a relatively new proposal of a workow/business processing system, that supports a concise and powerful workow language and handles complex data, transformations and Web service integration. YAWL extends Petri nets by introducing some workow patterns (for multiple instances, complex synchronisations, and cancellation) that cannot be easily expressed using (high-level) Petri nets. Petri net based tools such as [17], and YAWL-based tools such as [16] can be employed to formally analyse YAWL workows (viz., verify properties such as reachability, soundness, or lock-freedom) [13]. Furthermore, not being a commercial language, YAWL supporting tools (editor, engine) are freely available (`www.yawl-system.org`). More details on YAWL will be given in Subsection 3.2.

[3]Note that this is a high-level notation. SMoL actually employs an XML Schema based on the schema defined by BPEL 2.0.

```
COM ::=
    Sequence COM+ EndSequence |
    Flow COM+ EndFlow |
    While boolCond COM EndWhile |
    RepeatUntil boolCond COM EndRepeatUntil |
    If boolCond COM Else COM EndFlow |
    Assign [Copy FROM TO EndCopy]+ EndFlow |
    Pick [pickGuard COM]+ EndPick |
    InformationHandler
            COM [infoGuard COM]+
    EndInformationHandler |
    FaultHandler COM [catchGuard COM]+ EndFaultHandler
boolCond    ::= logicalExpression
FROM        ::= variable | expression | literal | opaque
TO          ::= variable
pickGuard   ::= receiveMessage(gId?,opName) |
                receiveEvent(groupId?, evName) |
                wait(for?, until?)
infoGuard   ::= receiveMessage(gId?,opName) |
                receiveEvent(groupId?, evName) |
                wait(for?, until?, repeatEvery?)
catchGuard  ::= catch(fName) | catchAll()
```

Figure 3. SMoL Structured Commands.

3.2 The Core of SMoL2YAWL

SMoL2YAWL is a pattern-based compositional translator of SMoL programs into YAWL workflows. In the following we describe the YAWL patterns used by SMoL2YAWL for the translation of both basic and structured SMoL commands.

3.2.1 Basic Command Patterns

Each SMoL basic command (but primitives) translates to a single atomic YAWL task. (Intuitively, YAWL tasks correspond to Petri net transitions.) Figure 4 illustrates the patterns of the SMoL basic commands previously defined in Subsection 2.2.1.

For example, the **Empty** pattern consists of a dummy task with no inputs and outputs, while **Wait** consists of a task that inputs the duration of the timeout. Furthermore, in order to terminate the execution of the workflow translating a SMoL program, **Exit** is to be connected to the output condition of the workflow (see Subsection 3.2.2). (Intuitively, YAWL conditions correspond to Petri net places.) The can-

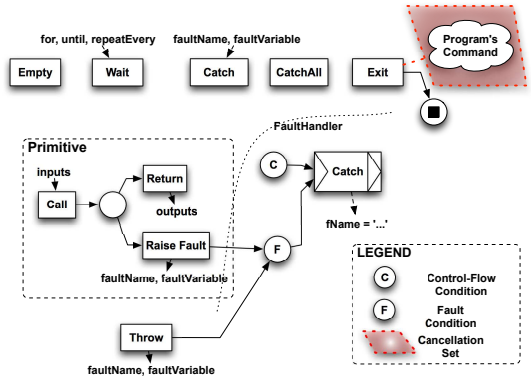

Figure 4. Basic Command Patterns.

cellation set[4] of **Exit** makes sure there are not tokens left in the workflow in case of an abnormal execution of the workflow.

Each SMEPP primitive translates to a *Call* task linked through a deferred choice to two other tasks – *Return* and *RaiseFault*. *Call* models the start of the primitive's execution, and it inputs the primitive's inputs. Its execution places a token in the deferred choice condition. Then, the environment (viz., the client of the workflow) decides whether the primitive terminates successfully – by executing the *Return* task, or whether the primitive raises a fault – by executing the *RaiseFault* task. On the one hand, *Return* has to forward a token to the pattern of the command whose execution follows the primitive in the SMoL program, as defined by their parent pattern (see Subsection 3.2.2 for details on how the structured patterns link their children patterns). On the other hand, *RaiseFault* signals a fault to the parent **FaultHandler** (if any, otherwise to the default one) by forwarding a token to the fault condition of the enclosing **FaultHandler** (again, see Subsection 3.2.2 for details on the **FaultHandler** pattern).

3.2.2 Structured Command Patterns

In this subsection we describe the patterns translating the SMoL structured commands previously defined in Subsection 3.2. As one would expect, the patterns of the structured SMoL commands are compositionally constructed by suitably linking the patterns that translate the children activities of the structured commands.

Sequence and Assign. Figure 5 sketches the **Sequence** pattern. The dummy *Begin(Sequence)* and *End(Sequence)* tasks mark the beginning of the SMoL sequence, and its termination, respectively. Each child command of the SMoL sequence translates to a (possibly structured) pattern[5]. These patterns are linked sequentially, in lexical order of occurrence of their corresponding commands in the sequence. *Begin(Sequence)* is in charge of enabling for execution the pattern of the first command in the sequence. The termination of the first **Command** pattern enables for execution the second **Command** pattern, and so on, until the last **Command** pattern, whose execution enables *End(Sequence)*. The **Assign** pattern is quite similar; it simply employs atomic *Copy* tasks instead of child **Command** patterns. Each *Copy* maps an input *from* variable onto an output *to* variable, as in the corresponding SMoL copy. (Note that SMoL, similarly to BPEL and YAWL, uses XPath and XQuery for data manipulation.)

Figure 5. The Sequence pattern.

Flow. Figure 6 graphically depicts the **Flow** pattern. Similarly to previous structured patterns, it employs *Begin* and *End* tasks, which mark the initiation and the termination, respectively, of the flow command. Furthermore, each child command defined in the SMoL flow translates to a possibly structured **Command** pattern. *Begin(Flow)* employs an AND-split construct[6] so as to enable for execution all children **Command** patterns. Dually, *End(Flow)* employs an AND-join construct[7] so as to make sure that the **Flow** terminates only when all children **Command** patterns have finished their execution.

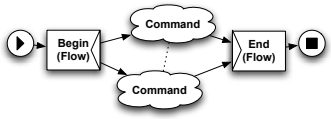

Figure 6. The Flow pattern.

While and RepeatUntil. Figure 7 illustrates the **While** pat-

[4] YAWL cancellation sets serve to remove tokens from a workflow. When a task is executed, all tokens from its cancellation set (if any) are removed.

[5] In the following figures, we will represent the pattern of a generic (basic or structured) command by a "cloud".

[6] YAWL tasks employ split constructs, which can be EMPTY, AND, OR, or XOR. Intuitively, the split construct of a task T species "how many" tasks are to be executed after T finishes. For example, in case of an AND-split, a token is generated on all the output links.

[7] YAWL tasks also employ join constructs, which can be again EMPTY, AND, OR, or XOR. Intuitively, the join of a task T species "how many" tasks before T are to be terminated in order to execute T. For example, a task with an AND-join can execute only after receiving at least one token on each of its input links.

tern. *Begin(While)* computes the value of the boolean guard condition, as defined in the SMoL while. *Begin(While)* employs a XOR-split[8] so as to enable for execution either the child **Command** pattern – if the guard holds true, or *End(While)* – otherwise. Similarly, *End(While)* recomputes the boolean guard and it re-enables for execution the child **Command** pattern – if the guard holds true, or it terminates the **While** – otherwise. Note that the **While** pattern imposes a slight modification to the child **Command** pattern. If the child **Command** is a basic pattern, then the respective task has to employ a XOR-join. Otherwise, if **Command** is a structured pattern, then its *Begin* task has to employ a XOR-join. The XOR-join makes sure that the **Command** pattern gets enabled either by *Begin(While)* – in the first cycle, or by *End(While)* – for subsequent cycles. The **RepeatUntil** pattern differs from **While** in that *Begin(RepeatUntil)* patterns simply forwards a token to the child **Command** pattern without checking any guard. Furthermore, *End(RepeatUntil)* forwards a token to the child **Command** provided the guard holds false.

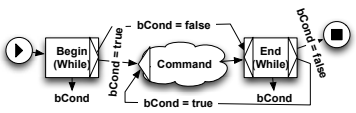

Figure 7. The While pattern.

If. Figure 8 presents the **If** pattern. *Begin(If)* computes the boolean guard defined by the SMoL if command to be translated, and it either enables for execution the *Command* pattern corresponding to the *then* branch – if the guard holds true, or the **Command** pattern corresponding to the *else* branch – if the guard holds false, otherwise. The termination of the selected **Command** enables for execution *End(If)*.

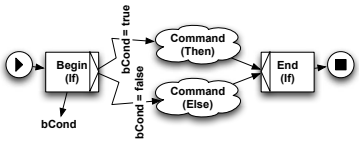

Figure 8. The If pattern.

Pick. Figure 9 depicts the **Pick** pattern. In short, each message or event branch consists of a **Receive** pattern linked to a **Command** pattern, corresponding to a message or event branch of the SMoL pick to be translated. Furthermore, each alarm branch consists of a **Wait** pattern linked to a **Command** pattern. *Begin(Pick)* enables from the control-flow viewpoint all the **Receive** and **Wait** patterns. The execution of a **Receive** (by the environment) corresponds to receiving a message or an event on the respective branch. Consequently, the **Receive** cancels all control-flow tokens of the other message and event branches, as well as it cancels all the alarms. Furthermore, the execution of the **Pick** continues with the **Command** of the selected message branch. On the other hand, the termination of a **Wait** cancels the control-flow tokens of all message and event branches, as well as it cancels all other alarms. Then, the **Pick** continues by executing the **Command** of the selected alarm branch. The termination of the selected message or alarm branch enables *End(Pick)*, which marks the end of the **Pick**.

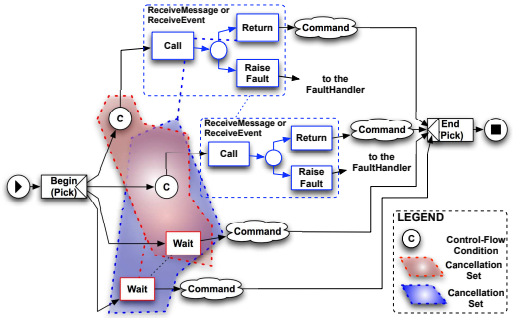

Figure 9. The Pick pattern.

InformationHandler. Figure 10 represents the **InformationHandler** pattern. **InformationHandler** is somewhat similar to **Pick**. *Begin(InformationHandler)* enables the **Receive** and **Wait** patterns, whose execution roughly corresponds to selecting a message/event, or alarm branch, respectively. The execution of a **Receive** enables, on the one hand, the **Command** of the respective message or event branch, and on the other hand, it re-enables the **Receive**. While the former starts the execution of the branch **Command**, the latter gives the possibility of re-executing the same branch. The execution of a **Wait** enables the **Command** of the respective alarm branch, as well as the *RepeatEvery* task (if any). Furthermore, each completion of the *RepeatEvery* task leads to re-executing the branch **Command**.

Begin(InformationHandler) also enables the **InformationHandler**'s **Command**. The completion of the latter leads to cancelling all control-flow tokens that enable message and event branches, as well as all running alarms. In this way, no other message, event or alarm branches can be selected for execution. However, the **Command** patterns of

[8]Roughly, tasks with XOR-splits employ predicates (viz., logical expression) that guard the output links. Only one token is to be sent on the predicate holding true.

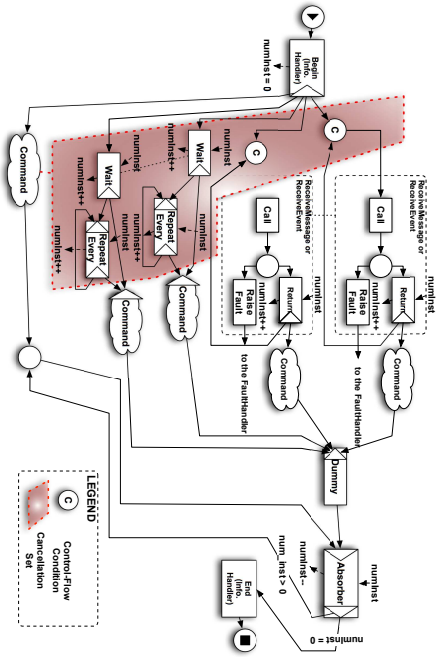

Figure 10. The InformationHandler pattern.

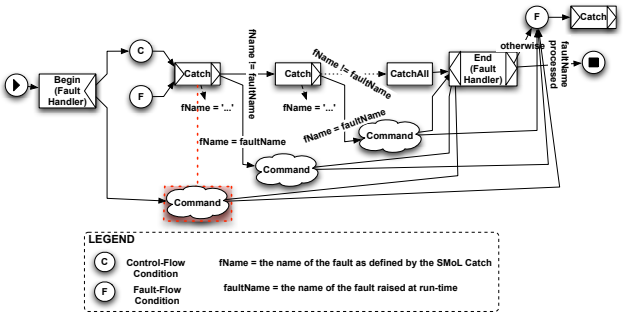

Figure 11. The FaultHandler pattern.

branches selected for execution prior to the completion of the **InformationHandler**'s **Command** are allowed to terminate. This is achieved through the use of variables, and through the cycle involving the *Absorber* task and the deferred choice condition. On the one hand, each selection of a branch increments a *numInst* variable, which is initially set to *0*. On the other hand, the termination of the **InformationHandler**'s **Command** places a token in the deferred choice, input of the *Absorber* task. Then, each completion of a branch **Command** enables the *Dummy* task and further *Absorber*, which now has both its input tokens. The *Absorber* task either places a token back into its input condition – if there are branch *Command*s still to be completed (viz., *numInstances* > 0), or it enables the *End(InformationHandler)* task – if all running branch *Command*s have completed their execution (viz., *numInst = 0*). The execution of the *End(InformationHandler)* marks the termination of the **InformationHandler** pattern.

FaultHandler. All SMEPP primitives may raise exceptions (e.g., invalidGroupId, accessDenied, or callerNotInGroup). However, due to space limitations we did not represent them in Figure 1. For details on the SMEPP exceptions please see [15].

Figure 11 depicts the **FaultHandler** pattern. *Begin(FaultHandler)* enables the **FaultHandler**'s **Command**, as well as, it also enables from the control-flow viewpoint the first *Catch*/*CatchAll* task. The *Catch*/*CatchAll* tasks are linked sequentially, in lexical order of occurrence of their corresponding catch/catchAll elements in the SMoL faultHandler to be translated. The **FaultHandler**'s *Catch*/*CatchAll* can only be executed upon the reception of a fault. In order to model such behaviour, the first *Catch*/*CatchAll* task also inputs a flow condition. The cancellation set (associated to the first *Catch*/*CatchAll* in the **FaultHandler**) interrupts the execution of the main command when a fault is being raised inside the **FaultHandler**.

We assume that the erroneous execution of the **FaultHandler**'s **Command** places a token in the fault condition of the **FaultHandler**. Consequently, the first *Catch*/*CatchAll* task executes. If the name of the fault defined by the respective *Catch* (viz., *fName* in the figure) matches the name of the fault raised by the **FaultHandler**'s **Command**, then the execution of the **FaultHandler** continues with the **Command** of the respective catch branch. Otherwise, its execution continues with the next *Catch*/*CatchAll* task. Note that a *CatchAll* task matches all faults. It is also important to note that if a fault raised by the **FaultHandler**'s **Command** cannot be processed by the **FaultHandler**; in this case, *End(FaultHandler)* is in charge of forwarding the fault to the enclosing **FaultHandler** (if any, or to the default **FaultHandler** associated to the entire SMoL program). The scenario is similar if the execution of a *Catch*/*CatchAll* raises a fault.

4 Example

Consider the SMoL code in Figure 12 describing a monitoring service (MS). MS first waits to receive from its client the id of a group in which it has to monitor the ambient temperature. Then, it subscribes to the "temp10s" event raised in this group[9], and it waits to receive events during one hour, after which it unsubscribes from the monitored event.

[9]For space limitations we do not present here the SMoL code of the entities raising "temp10s" events.

Finally, it signals the termination of the monitoring period to its client. Figure 13 depicts the YAWL workflow corresponding to MS[10]. One may note the default **FaultHandler** that catches faults raised by the service. One may also note two cancellation sets: one in charge of aborting the execution of the workflow (see *CatchAll*), and another one that disables the reception of events in the *InformationHandler* (see *EndSequence*).

```
Sequence
  <ownerPeer, gid> = receiveMessage("monitor")
  subscribe("temp10s", gid)
  InformationHandler
   Sequence
    wait("PT1H")
    unsubscribe("temp10s", gid)
   EndSequence

   <cId, temp> = receiveEvent(gid, "temp10s")
      // Use temp for something
  EndInformationHandler
  reply(ownerPeer, "monitor")
EndSequence
```

Figure 12. Monitoring Service (SMoL).

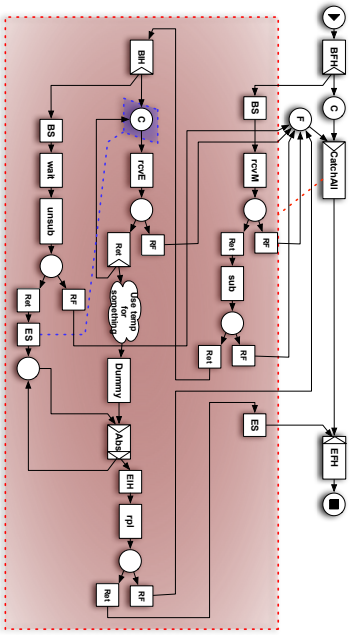

Figure 13. Monitoring Service (YAWL).

[10]The figure uses abbreviated task names and it does not display task inputs and outputs.

5 Concluding Remarks

In this paper we have briefly presented a service-oriented model to specifiy the interaction among peers in P2P systems, thus hiding low-level details concerning the supporting infrastructure. The core concepts of the model are:

- **The SMEPP primitives**, which allow peers to create, join, and leave groups, as well as to publish and unpublish services inside groups. Peers can either communicate directly with other peers, or they can invoke peer or group services. Furthermore, peers and services can communicate through event and fault notifications.
- **SMoL**, which is a language for the orchestration of SMEPP primitives into peer and service code, which employs state-of-the-art concepts for the definition of Web service behaviour. SMoL defines sequential, flow, (deterministic) conditional, iterative, and (non-deterministic) choice composition operators. Furthermore, it supports the definition of information (message and event) and fault handlers.

Various service and interaction models have been proposed for modelling EP2P systems. Some of them are inspired by the service-oriented architecture paradigm (e.g., [6, 11]), others are based on/extend JXTA [9] (e.g., [2, 3]), while others are data-driven coordination models (e.g., [7, 10]).

SOA-based P2P models. Gehlen and Pham [6] model peer interfaces to the distributed environment through SOAP components, which serve for exchanging, encrypting and marshalling SOAP messages. Their approach employs local and remote registries to store WSDL descriptions of the services deployed in the framework, and remote services, respectively. Maheshwari et al. [11] propose a service model based on a message queue cluster that intercepts and delivers SOAP messages exchanged by peers (Web services) so as to achieve high scalability, availability, fault tolerance, and load balancing. The main downside of such approaches is that they do not offer a high-level API to be used by developers for rapid application prototyping. Furthermore, the use of SOAP and Web service technologies makes these approaches unusable on low-end devices (e.g., PDAs or smartphones).

JXTA-based P2P models. Alda and Cremers [2] describe DeEvolve, a P2P architecture based on Juxtapose (JXTA [9]). DeEvolve introduces two languages: CAT – for expressing peer services as compositions of components, and PeerCAT – for expressing compositions of peer services. A main feature of DeEvolve is that PeerCAT can define exception handlers to cope with peer failures. Bisignano et al. [3] introduce JMobiPeer, a P2P computing platform developed on top of JXTA. JMobiPeer defines modules for transport and service protocols, for peer and peer

group management, and for peer advertisement and discovery management. Similarly to JXTA, advertisements provide information of available services, peers and groups, as well as pipes and end points. These approaches, however, are tightly coupled to JXTA (protocols and implementation). Furthermore, the JXTA advertisements do not give developers the flexibility of defining rich service contracts that can be used for enhanced service discovery.

Data-driven coordination models. Handorean et al. [7] introduce *follow-me sessions* that express the interaction of a client with a service that is offered by several providers, in order to achieve a continuity of service provision. The paper discusses techniques for migrating processes between hosts, or partial results to alternate providers, for allowing temporary client disconnections while providers continue processing, and for letting clients use partial results until an alternate provider is found. Lucchi and Zavattaro [10] describe WSSecSpaces (W3S), Linda-based interaction model for Web services. The model allows for loosely-coupled Web services, in the way that a Web service can issue a request and then terminate. Then, the request is processed at a later time e.g., by a service that becomes online. These approaches however focus on continuous service provision, by decoupling service providers from their interfaces. They do not aim at providing a simple high-level API for the development of P2P specifications.

A thorough comparative analysis of related work in embedded peer-to-peer systems can be found in [14]. Roughly, we argue that existing service and interaction models generally either do not take into account key requirements (e.g., group-aware security, asynchronous, synchronous, and event-based communication, or service contracts), or they do not provide an abstract modelling language and API that pave the way for application prototyping and for simulating and verifying the behaviour of peers and services, and their interactions.

In this paper, we have also presented **SMoL2YAWL**, a pattern-based compositional translator of SMoL programs into YAWL workflows. On the one hand, SMoL2YAWL provides a lightweight semantics for SMoL programs, which can be used to simulate SMEPP entities (e.g., using the YAWL engine), as well as to analyse the behaviour of SMEPP entities by analysing their corresponding YAWL workflows [13].

In [5] we have defined an abstract semantics for a simple calculus over the SMEPP primitives. Such semantics formally establishes whether a set of SMEPP processes (viz., peer or service codes) can be executed together without locking, and it is currently exploited to develop a prototype MAUDE-based analyser. As we already mentioned, we have also developed a prototype tool for the transformation of SMoL descriptions into Java code. Our future work will be devoted to engineering our proof-of-concept implementation of SMoL2YAWL, and to integrating it with YAWL-based simulation and analysis tools, and with the SMoL2Java prototype. Our more long-term objective is the development, in the context of the SMEPP project, of a full-fledged SMEPP middleware based on the service model described in this paper.

References

[1] M. Albano, A. Brogi, R. Popescu, M. Diaz, and J. Dianes. Towards secure middleware for embedded peer-to-peer systems: Objectives and requirements. In *Proceedings of RSPSI'07*, 2007. `http://www.igd.fhg.de/igd-a1/RSPSI2/papers/Ubicomp2007_RSPSI2_Albano.pdf`.

[2] S. Alda and A. Cremers. Towards composition management for component-based peer-to-peer architectures. *ENTCS*, 114:47–64, 2005.

[3] M. Bisignano, G. D. Modica, and O. Tomarchio. JMobiPeer: A middleware for mobile peer-to-peer computing in manets. In *ICDCS'05*, pages 785–791, 2005.

[4] A. Brogi and R. Popescu. From BPEL processes to YAWL workflows. In M. Bravetti, M. Nunez, and G. Zavattaro, editors, *Proceedings of WS-FM'06, LNCS*, volume 4184, pages 107–122, 2006.

[5] A. Brogi, R. Popescu, F. Gutierrez, P. Lopez, and E. Pimentel. A service-oriented model for embedded peer-to-peer systems. In *Proceedings of FOCLASA'07, ENTCS. To appear*, 2007.

[6] G. Gehlen and L. Pham. Mobile web services for peer-to-peer applications. In *Proceedings of CCNC'05*, pages 427–433, 2005.

[7] R. Handorean, R. Sen, G. Hackmann, and G.-C. Roman. Supporting predictable service provision in manets via context aware session management. *JWSR*, (3):1–26, 2006.

[8] JXTA. `http://www.jxta.org/`.

[9] JXTA homepage. `http://www.jxta.org/`.

[10] R. Lucchi and G. Zavattaro. WSSecSpaces: a secure data-driven coordination service for Web services applications. In *SAC'04*, pages 487–491. ACM, 2004.

[11] P. Maheshwari, S. Kanhere, and N. Parameswaran. Service-oriented middleware for peer-to-peer computing. In *INDIN'05*, pages 98–103, 2005.

[12] OASIS. BPEL v2.0. `http://www.oasis-open.org/committees/download.php/23974/wsbpel-v2.0-primer.pdf`.

[13] R. Popescu. *Aggregation and Adaptation of Web Services*. VDM Verlag Dr. Müller, 2008. ISBN: 978-3-8364-6280-8.

[14] SMEPP Coalition. D1.1: State of the art and generic middleware requirements. `http://www.smepp.org/`.

[15] SMEPP Coalition. D1.2: Security requirements of ep2p applications. `http://www.smepp.org/`.

[16] E. Verbeek. WofYAWL V0.3. `http://home.tm.tue.nl/hverbeek/wofyawl03.pdf`.

[17] E. Verbeek and W. van der Aalst. Woflan 2.0: A petri-net-based workflow diagnosis tool. In *LNCS*, volume 1825, pages 475–484, 2000.