A Formalization of the SMEPP Model in Maude*

Francisco Durán
University of Málaga
duran@lcc.uma.es

Francisco Gutiérrez
University of Málaga
pacog@lcc.uma.es

Pablo López
University of Málaga
lopez@lcc.uma.es

Ernesto Pimentel
University of Málaga
ernesto@lcc.uma.es

ABSTRACT

This paper introduces a service-oriented model for the description of embedded Peer-to-Peer (EP2P) systems and formalizes the proposed model in Maude. The model is organized around the notions of *groups* of peers and *services* offered by these groups. We first summarize the main concepts of the model and then present αSMoL, an abstract language with a formal semantics that provides a solid ground to develop tools for the automated analysis and verification of EP2P specifications. We then describe a formalization of αSMoL in Maude, and introduce an example to illustrate both the expressive power of the model and the possibilities of Maude to support automated verification of properties of αSMoL programs.

Keywords

Peer-to-Peer Systems, Service-Oriented Models, Formal Semantics, Automated Verification

1. INTRODUCTION

Peer-to-Peer (P2P) systems are distributed computing systems where all network elements act both as service consumers (clients) and service providers, frequently communicated over MANETs [12]. Embedded Peer-to-Peer (EP2P) systems [5] are P2P systems where small, low-powered, low-cost embedded devices cooperate in exchanging and processing information using wireless channels. EP2P systems pose new challenges in the development of software for distributed systems, such as decentralisation, unreliable communication links, and dynamic topologies.

To overcome the aforementioned implementation challenges, it is convenient to define a middleware (including formal tools and methodologies) to abstract away from details of the underlying infrastructure. Developing such a middleware is a demanding task, since a number of critical requirements, such as mobility, security concerns, or service discovery must be met.

The Secure Middleware for Embedded Peer-To-Peer Systems (SMEPP) European Project (FP6-IST-033563) is devoted to develop a middleware that must be secure, highly customisable, and amenable to be deployed in a wide range of devices (from PDAs to embedded sensors) and domains (from mission critical systems to consumer entertainment). SMEPP provides a high-level, service-oriented model to program the interaction of peer-to-peer applications, regardless low-level communication details.

The operational semantics of the SMEPP is provided by the αSMoL calculus, which has been presented by Brogi et al. in [2]. The semantics of αSMoL allows to establish whether a program (a set of peers and services) can be concurrently executed without locking, and provides a solid ground to develop tools for the automated analysis and verification of EP2P systems. To automate the formal analysis on αSMoL programs, we have started using the Maude system and its formal environment. We present here a formalization of αSMoL in Maude. This Maude specification is executable, thus providing a prototype implementation of αSMoL programs. Moreover, Maude's formal environment may help us to automate the formal analysis of αSMoL programs. Maude's execution and searching capabilities, its model checker, its theorem prover, its termination, Church-Rosser and confluence checkers, etc. can be of great help.

This paper is organized as follows. First we briefly summarize the main concepts of the SMEPP model. In Section 3 we present αSMoL. Section 4 discusses our formalization of αSMoL in Maude. Section 5 introduces a simple αSMoL program and its encoding in Maude, and gives some hints on the kind of analysis and verification support provided by the Maude tools. We conclude with some remarks and ideas about future work.

2. THE SMEPP MODEL

The SMEPP service-oriented model revolves around a number of key concepts:

Services have contracts and implementations. The contract describes the service, while the implementation is the executable service (e.g., a Java service). Services expose *ser-*

*This research has been partially supported by the European Project SMEPP FP6-IST-033563 and by the PICaSSo project (P06-TIC2250) funded by the Regional Government of Andalusia.

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3916
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3916

vices operations in their contracts. Service operations can be invoked and provide a tight way of communication, in either one or two-ways (request-response) similarly to WSDL [14].

Peers contain exactly one peer program, and may offer a number of services.

Groups are the basic abstraction on which service offers and requests are organized. A group is a set of peers that share services or provide services to other peers in the group. Services are offered by groups and, for a peer to use a service, it must be a member of the group that offers the service.

Events model notifications and provide a loose way of communication. Entities can raise an event and resume processing as soon as the event is dispatched to the middleware. In addition, entities can wait for events and resume as soon as they receive such event notifications.

Exceptions can be raised by the middleware to signal failures, privilege violations, runtime errors, etc. Besides, a service provider can also raise exceptions to a requester to signal errors in the service request or execution.

The SMEPP model defines a set of abstract primitives to deal with the concepts outlined above. These primitives provide the basic building blocks to specify the interaction in a EP2P system (e.g, service publication and invocation). The fundamental SMEPP primitives are carefully formalized in αSMoL, as shown in the next section.

3. αSMoL: A CALCULUS FOR SMEPP

To define a calculus for the SMEPP primitives and the corresponding semantics, we need to introduce some preliminary concepts. Let $\mathcal{P}$ and $\mathcal{S}$ be the sets of peer and service identifiers, respectively. We will also consider a set of group identifiers, $\mathcal{G}$, including a special symbol 0 which will be called the *universal* group. To refer to entities we will use triples $(g, p, s) \in \mathcal{G} \times \mathcal{P} \times \mathcal{S}$ called middleware uniform resource locators (locators for short). A locator will be written as $g.p.s$ and addresses a service s contained in a peer p running in a group g. When g and s are 0 the locator denotes the peer code; that is, the peer code is a special service (denoted by 0) running in the universal group (0). The set of locators is denoted by $MURL$.

As it was previously mentioned, services will be characterized by a contract service and a set of operations. We will denote by $\mathcal{C}$ and $\mathcal{OP}$ the sets of contract services and operations signatures, respectively. We will denote by $C^{\emptyset}$ the empty contract. The way in which these sets are defined is not relevant for our purposes, although usually they will be given by XML-like specifications. Finally, $\mathcal{EV}$ and $\mathcal{EX}$ denote the sets of event and exception names, respectively.

A group is represented by a labelled triple $\langle P, Sr, Sb\rangle_g$ consisting of a set $P \subseteq \mathcal{P}$ of peer identifiers (the *members* of the group), a set Sr of service identifiers (the *services provided* by the group), and a set Sb of subscriptions of entities to events provided by other services in the group. The set of services running in the group is a set of tuples, i.e., $Sr \subseteq \mathcal{S}\times\mathcal{C}\times\mathcal{P}$, where the first component is the service identifier, the second one is the contract exposed by the service, and the last one is the identifier of the peer providing the service. The set of subscriptions to events is a set of triples, i.e., $Sb \subseteq MURL \times MURL \times \mathcal{EV}$, where the first component denotes the subscribed entity, the second one is the entity to which it is subscribed, and the third component represents the signature of the subscribed event.

The αSMoL primitives below are an abstraction of the SMEPP primitives and provide a means to specify service-oriented interaction in an EP2P system:

$$
\begin{array}{rcl}
a & ::= & m = \mathsf{getId}() \mid g = \mathsf{createGroup}() \mid \mathsf{joinGroup}(g) \\
& \mid & \mathsf{leaveGroup}(g) \mid m = \mathsf{getServices}(m, sc) \\
& \mid & s = \mathsf{publish}(g, sc) \mid \mathsf{unpublish}(m) \mid \mathsf{throw}(ex, ?in) \\
& \mid & \mathsf{subscribe}(ev, g) \mid \mathsf{unsubscribe}(ev, g) \\
& \mid & res = \mathsf{invoke}(m, op, ?in) \mid \mathsf{reply}(m, op, out) \\
& \mid & \langle m, ?res\rangle = \mathsf{receiveMessage}(g, op) \\
& \mid & \mathsf{send}(m, op, ?in) \mid \langle ?m, ?res\rangle = \mathsf{receive}(g, op) \\
& \mid & \mathsf{event}(g, ev, ?in) \mid \langle ?m, ?res\rangle = \mathsf{receiveEvent}(g, ev)
\end{array}
$$

where $g \in \mathcal{G}$ stands for *group identifier*, $s \in \mathcal{S}$ for *service identifier*, $sc \in \mathcal{C}$ for *service contract*, $m \in MURL$ for locator, $op \in \mathcal{OP}$ for *operation*, $ev \in \mathcal{EV}$ for *event*, $ex \in \mathcal{EX}$ for *exception* name, *in* for *input*, *out* for *output*, and res for the *result* returned by a primitive. Parameters prefixed with ? are optional.

Recall that a peer contains exactly one peer program (peer code), and may contain one or more service programs (service code). To define αSMoL programs we consider a set of compositional operators, usual in concurrent languages. In particular, we include prefix actions (.), parallel composition ($\|$), non-deterministic choice (+ or $\sum$), exception handling (@), and local definitions (D):

$$
\begin{array}{rcl}
\mathsf{Prg} & ::= & x = \mathsf{new}().\mathsf{Code} \\
\mathsf{Code} & ::= & \mathsf{0} \mid a.\mathsf{Code} \mid \mathsf{Code} \parallel \mathsf{Code} \mid \mathsf{Code@Excs} \\
& \mid & \mathsf{Choice} \mid D(\vec{x}) \\
\mathsf{Excs} & ::= & [] \mid (\sum_i \mathsf{catch}(ex_i, in_i).\mathsf{Code}_i) : \mathsf{Excs} \\
\mathsf{Choice} & ::= & \sum_i a_i.\mathsf{Code}_i
\end{array}
$$

where 0 is the empty program, a and a_i denote αSMoL primitives, ex_i stands for an exception name, in_i for the arguments of the exception, and $D(\vec{x})$ represents a program definition of the form $D(\vec{y}) \stackrel{def}{=} \mathsf{Code}$, where $\vec{y}$ denotes a sequence of arguments.

The αSMoL semantics must obviously deal with peers before and after they register in the middleware. To that end we extend αSMoL programs by introducing the notion of labelled code ($[\mathsf{Code}]_{MURL}$). αSMoL programs are then given by the following syntax::

$$
\mathsf{Prg}^* \quad ::= \quad \mathsf{Prg} \mid [\mathsf{Code}]_{MURL}
$$

thus non-labelled code (Prg) is used for programs running in peers not yet registered in the middleware —in this case the first action must be the execution of new to register the peer— while labelled code ($[\mathsf{Code}]_{g.p.s}$) denotes programs executed by peers already registered in the middleware (and therefore having a unique identifier, p).

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3916
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3916

To define the semantics of αSMoL programs we introduce judgements of the form:

$$\Gamma \rightsquigarrow \Pi$$

where Γ is a set of groups called the *environment* and Π is a set of αSMoL programs (Prg^*).

The semantics is described by means of inference rules of the form:

$$\frac{Pr \qquad \Gamma \rightsquigarrow \Pi}{\Gamma' \rightsquigarrow \Pi'} \qquad \text{(Name)}$$

where the judgement above the line is a hypothesis, the one below the line a conclusion, and Pr a proviso.

For every construct of αSMoL —primitives and composition operators— we have inference rules describing its semantics. For example, the parallel composition operator is defined by the rule:

$$\frac{\Gamma \rightsquigarrow [A@Es]_{g.p.s}, [B@Es]_{g.p.s}, \Pi}{\Gamma \rightsquigarrow [(A \parallel B)@Es]_{g.p.s}, \Pi} \qquad \text{(Parallel)}$$

In addition, we shall have an axiom to indicate the termination of an execution:

$$\frac{}{\Gamma \rightsquigarrow} \qquad \text{(Terminate)}$$

stating that the empty program terminates in any "legal" environment Γ. A (successful) finite execution is a sequence of judgements whose first element is a termination judgement and such that every other judgement is the conclusion of the previous one.

4. A REWRITING LOGIC SEMANTICS OF αSMoL

Maude [3, 4] is a high-level language and a high-performance system in the OBJ algebraic specification family. It supports membership equational logic [1] and rewriting logic [11] specification and programming of systems. Thus, Maude integrates an equational style of functional programming with rewriting logic computation. We informally describe in this section those Maude's features necessary for understanding the paper; the interested reader is referred to [4] for additional details.

Rewriting logic has been shown to be a very good logical and semantic framework [7] in which a wide range of systems, including models of concurrency, distributed algorithms, network protocols, semantics of programming languages, etc. can be expressed and given semantics (see [8, 4] for an overview of some application areas of rewriting logic and Maude, with pointers to papers and web sites where more information can be found). Given its good representational features, rewriting logic can be used as a universal logic which can faithfully express the inference systems of many other logics. Moreover, as a semantic framework, it can be used to formally specify different systems as rewrite theories. Since those specifications usually can be executed in Maude, they in fact become interpreters for such systems, and allow a variety of formal analyses for them.

Rewriting logic is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. The models of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ capture the intuitive idea of a *rewrite system*, in the sense that they are systems whose states are E-equivalence classes of terms, and whose transitions are concurrent rewrites using the rules in R. By adopting a logical instead of a computational perspective, we can alternatively view such models as *logical systems* in which formulas are rewritten to other formulas by concurrent rewrites, which correspond to proofs for such a logic.

There has been a broad effort exploring the features of rewriting logic, and in particular of Maude, as a logical and semantic framework for representing and executing inference systems (see, e.g., [13, 10]). The basic idea is to represent judgments in the inference system as terms in Maude, and inference rules as rewrite rules. A very general possibility is to map an inference rule of the form

$$\frac{S_1 \ldots S_n}{S_0}$$

into a rewrite rule of the form $S_1 \ldots S_n \longrightarrow S_0$ that rewrites multisets of judgements S_i. This mapping is correct from an abstract point of view [7]. However, thinking in terms of executability of the rewrite rules, it is more appropriate to consider rewrite rules of the form $S_0 \longrightarrow S_1 \ldots S_n$, which still rewrite multisets of judgements but go from the conclusion to the premises. Rewriting with these rules corresponds to searching for a proof in a bottom-up way. Again, this mapping is correct, and, as before, the intuitive idea is that the rewriting relation corresponds to the horizontal bar separating conclusion from premises in the typical presentation of inference rules.

Assuming appropriate definitions (see below), a rule like, e.g., the Parallel one presented in Section 3, can be represented as a rewrite rule as:[1]

```
rl [PARALLEL] :
        Gamma ~> [(A || B) @ Es] g . p . s, Pi
=> ------------------------------------------------
   Gamma ~> [A @ Es] g . p . s, [B @ Es] g . p . s, Pi
.
```

Note that the "representation distance" is very small, we have mirrored very closely the constructions introduced in the αSMoL calculus.

The following are the main declarations making this possible:

- Sorts `GId`, `PId`, and `SId` of group, peer, and service identifiers, respectively.

[1] In Maude, the text written after '`---`' is considered as a comment.

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3916
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3916

- A `MURL` sort of resource locators, with a constructor

```
op _._._ : GId PId SId -> MURL .
```

  to represent a locator $g.p.s$ as a term `g . p . s` for variables `g`, `p`, and `s` of sorts `GId`, `PId`, and `SId`, respectively.

- Sorts `SC`, `OP`, `EV`, and `EX` for service contracts, and operation, event, and exception names, respectively.

- A `Primitive` sort of αSMoL primitives with the following operators:

```
op _= getId() : MURL -> Primitive .
op _= createGroup() : GId -> Primitive .
op joinGroup(_) : GId -> Primitive .
op leaveGroup(_) : GId -> Primitive .
```

  Operators for the rest of primitives in Section 3 are declared similarly. Primitives with optional parameters require additional declarations.

- The declarations defining sorts `Code` and `Prg` of peer code and service code, respectively, are:

```
op _= new() ._ : PId Code -> Prg .

subsort Choice < Code .
op _._ : Primitive Code -> Choice [strat (1 0)] .
op _+_ : Choice Choice -> Choice [assoc comm] .
op 0 : -> Code .
op _||_ : Code Code -> Code [assoc comm] .
op _@_ : Code Excs -> Code .

subsort Catch < CatchChoice < Exs .
op catch(_,_)._ : EX Data Code -> Catch .
op _+_ : CatchChoice CatchChoice -> CatchChoice
    [assoc comm] .
op [] : -> Excs .
op _:_ : CatchChoice Excs -> Excs .
```

  where `0` represents the empty program, `_||_` parallel composition, `_+_` guarded non-deterministic choice, and `_@_` exception handling.

- The `Prog*` sort of labelled code and unregistered programs includes the following declarations.

```
subsort Prg < Prg* .
subsort Code < Code* .
op dispatch(_,_,_) : MURL EV Data -> Code* .
op _= suspend(_,_) : PId OP MURL -> Code* .
op [_]_ : Code* MURL -> Prg* .
```

- The sort `Group` of groups has the following operator as constructor.

```
op <_,_,_>_ : Set{PId} Set{Service}
     Set{Subscription} GId -> Group .
```

  where `Set{PId}`, `Set{Service}`, and `Set{Subscription}` represent, respectively, sets of peer identifiers (the members of the group), service descriptions (services provided by the group), and event subscriptions (subscriptions to events provided by the services in the group).

  A service description includes a service identifier, the contract exposed by the service, and the identifier of the peer providing the service:

```
op ((_,_,_)) : SId SC PId -> Service .
```

  A subscription includes the subscribed entity (first component), the entity to which it is subscribed, and the name of the subscribed event:

```
op [_,_,_] : MURL MURL EV -> Subscription .
```

- Judgements of the form $\Gamma \rightsquigarrow \Pi$ are represented as terms of sort `Judgment`.

```
op _~>_ : Set{Group} Set{Prg*} -> Judgment .
```

- We also include declarations for
  - partial order relations on sets of group, peer, and service identifiers,
  - a partial order relation on locators,
  - removal operators for sets of services, subscriptions, and programs,
  - etcetera.

5. AN APPLICATION EXAMPLE

Given the specification of the αSMoL calculus in Maude as described above, we can write P2P applications as, e.g., the one presented in [2]. In that example we consider a provider of temperatures `TempServ`, that keeps reading temperatures from some external device and communicating these readings by dispatching `"Temperature"` events to the subscribers. This service is published, stopped, and unpublished by the peer code `TempPeer`. On the other hand, the service `TempServInkoker` keeps handling `"Temperature"` events and summing up the temperature readings. Upon the reception of the `"GetAverage"` operation, `TempServInvoker` computes the average temperature and sends back the result to the invoker. This service is published, started, stopped, and unpublished by the peer code `PeerClient`.

```
subsort String < Data .
ops TempPeer PeerClient : -> Prg .
ops TempServ TempServInvoker : -> Code .
op  AverageTemp : String String MURL -> Code .
ops TempServSC TempServInvokerSC : -> SC .
op  invalidGroupId : -> EX .
ops p q : -> PId .
op  g : -> GId .
ops s s' : -> SId .
op  avg t in : -> String . --- data
op  m : -> MURL .

eq TempPeer
  = ( p = new() .
      g = createGroup() .
      s = publish(g, TempServSC) .
      --- Processing other tasks
      send(g . p . s, "Terminate", "") .
      unpublish(g . p . s) .
      0 ) .
eq TempServ
  =   ( --- temp = 30
        event("Temperature", "30") .
        TempServ )
    + ( < m, in > = receiveMessage("Terminate") .
        0 ) .
eq PeerClient
```

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3916
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3916

```
  = ( p = new() .
      g . q . s = getServices(* . * . *, TempServSC) .
      joinGroup(g) .
      s' = publish(g, TempServInvokerSC) .
      send(g . p . s', "Start", g) .
      ---- Processing other tasks
      avg = invoke(g . p . s', "GetAverage", "") .
      send(g . p . s', "Terminate", "") .
      unpublish(g . p . s') .
      ---- Processing the average avg
      0
      @ catch(invalidGroupId, "service doesn't exist") ) .
eq TempServInvoker
  = ( < m, in > = receiveMessage(*, "Start") .
      subscribe("Temperature", m) .
      AverageTemp("", "", m) ) .
eq AverageTemp(a, n, g)
  =   ( < m, t > = receiveEvent(*, "Temperature") .
        ---- a' = a + t, n' = n + 1
        ---- AverageTemp(a + t, i + "1", m) )
        AverageTemp(a', n', g) )
    + ( < m, in > = receiveMessage(*, "GetAverage") .
        ---- avg = n == 0 ? ERROR : a/n
        reply(m, "GetAverage", a + "/" + n) .
        AverageTemp(a, n, g) )
    + ( < m, in > = receiveMessage(*, "Terminate") .
        unsubscribe("Temperature", m) .
        0 ) .
```

Note that once the appropriate identifiers are declared as constants, the programs and peers can be given as equations. Note that the syntax used in the same as in[2].

Once the system specifications are written using the modeling approach presented above, what we get is a rewriting logic specification of the system. Since the rewriting logic specifications produced are executable, this specification can be used as a prototype of the system, which allows us to simulate and analyze it.

Maude offers tool support for (controlled) execution, reachability analysis, model checking, termination and confluence checks, etc. Here we give a very simple example of the executing and searching capabilities of Maude on this particular example. Detailed descriptions of the commands and tools used here can be found in [4].

5.1 Controlled Execution

Given a Maude specification as the one described in the previous sections, it can be executed by just successively applying equations and rewrite rules on an initial term. Maude provides two different rewrite commands, namely `rewrite` and `frewrite`, which implement two different execution strategies, a top-down rule-fair strategy, and a depth-first position-fair strategy, respectively [4].

The `rewrite` and `frewrite` commands explore a possible execution of different initial models. However, a rewrite system do not need to be Church-Rosser and terminating, and there might be many different execution paths. Although these commands are enough in many practical situations where an execution path is sufficient for testing executability, the user might be interested in exploring all possible execution paths from the starting model, a subset of these, or a specific one.

There are currently three ways of controlling the execution process in Maude, which, ordered from the more complex to the easiest to use, are: (1) Using the reflective capabilities of Maude, with built-in metalevel functions such as `metaReduce`, `metaApply`, etc., which provide absolute control over the execution process [4]; (2) using invariant-driven strategies for executing specifications complying with given invariants [6]; and (3) using the Maude strategy language [9] to define strategy expressions that control the way terms are rewritten.

The following is a rewriting with the `frewrite` command:

```
frew < mt, mt, mt > 0g ~> TempPeer, PeerClient .
result Judgment:
    < pId(0), mt, mt > gId(1),
    < pId(0) U pId(4),
      (0s, C^0, pId(0)) U (0s, C^0, pId(4)),
      mt > 0g
   ~> [throw(invalidService, TempServCS) . 0]
        (0g . pId(4) . 0s)
```

The result given by the tool shows the term reached following one of the possible rewriting paths. In this case, an `invalidService` exception was raised and not handled.

Using Maude's strategy language (third alternative in the guiding options described above) in a very naive way, we can ask the engine to follow a specific path:

```
srew < mt, mt, mt > 0g ~> TempPeer, PeerClient
  using NEW ; NEW ; CREATE ; PUBLISH-1 .
```

The `srewrite` command gives as result all final states reached by applying the given strategy. In this case we have given just a sequence of rules, but other strategy combinators are available.

5.2 Reachability Analysis

Executing the system using the `rewrite` and `frewrite` commands means exploring just one possible behavior of the system. This can be partially improved by using rewriting strategies. However, it does not allow us to analyze the solutions found. The Maude `search` command allows us to explore (following a breadthfirst strategy up to a specified bound) the reachable state space in different ways.

We can look for terminal states as follows:

```
search [1] < mt, mt, mt > 0g ~> TempPeer, PeerClient
  =>* Gamma:Set{Group} ~> mt .
Solution 1 (state 13932)
Gamma:Set{Group} -->
  < pId(0) U pId(1), mt, mt > gId(2),
  < pId(0) U pId(1),
    (0s, C^0, pId(0)) U (0s, C^0, pId(1)),
    mt > 0g
```

The tool gives as result the states matching the given pattern. Note that each of the solutions given by Maude is

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3916
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3916

given as a substitution on the pattern state. Note also that we have limited the number of states to 1.

The shortest sequence of rewrites followed to reach a given state can be displayed by typing `show path` n, where n is the number of such state. If we are only interested in the labels of the applied rules, the command `show path labels` n can be used instead. In this case, `show path labels 13932` produces the sequence `NEW NEW CREATE ... UNPUBLISH NIL`, which is one of the possible paths leading to this state.

We can look for final states, either with an empty judgement or not as follows:

```
search [1] < mt, mt, mt > 0g ~> TempPeer, PeerClient
  =>! J:Judgment .
Solution 1 (state 181)
J:Judgment -->
  < pId(0), mt, mt > gId(2),
  < pId(0) U pId(1),
    (0s, C^0, pId(0)) U (0s, C^0, pId(1)),
    mt > 0g
  ~> [throw(invalidService, TempServSC) . 0]
        (0g . pId(1) . 0s)
```

Note that the first state matching the pattern is one in which appears an unhandled exception. Because of the limit on the number of states, this is the only one given as result.

We can search for states using any pattern, what would allow us to look for specific states, e.g., for deadlock states, or, in our example, for states with a `TempServInvoker` with a specific temperature. Moreover, conditions on the pattern states can also be specified, which gives us a great flexibility for expressing such searched states.

6. CONCLUDING REMARKS

We have presented a Maude specification of the αSMoL calculus, which provides an operational semantics of the SMEPP model. The SMEPP model is a high-level, service-oriented model to specify the interaction among peers, and also to define a simple semantics enabling the reasoning on abstract specifications of P2P systems.

We have illustrated how the executable Maude specification of the αSMoL calculus can be used as a prototypical implementation of the model. Moreover, Maude's formal environment allows us to check interesting properties about our models. We have illustrated how to execute sets of peers using Maude's built-in strategies, and also how to use some of its guiding facilities to control its execution. We have also shown how to use the searching facilities of Maude to check its termination, and to observe the form of the executions taking place.

In the future, we would like to use other tools in Maude's formal environment to further automate checks on our applications. Of particular interest is the use of its model checker. The type of applications considered, however, have the problem of having an infinite reachable state space. We hope that with equational abstractions we can be able to make interesting uses of it.

7. REFERENCES

[1] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1):35–132, 2000.

[2] A. Brogi, R. Popescu, F. Gutiérrez, P. López, and E. Pimentel. A service-oriented model for embedded peer-to-peer systems. In C. Canal, P. Poizat, and M. Viroli, eds., *6th Intl. Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2007)*, ENTCS 197, pp. 5–22, Lisbon, Portugal, Sept. 2007. Elsevier Science Publishers.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[5] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. The runes middleware: A reconfigurable component-based approach to networked embedded systems. In *Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05)*, Berlin (Germany), Sept. 2005.

[6] F. Durán, M. Roldán, and A. Vallecillo. Invariant-driven strategies for maude. *Electronic Notes in Theoretical Computer Science*, 124(2):17–28, 2005. S. Antoy and Y. Toyama, eds., 4th International Workshop on Reduction Stratategies and Programming (WRS'04).

[7] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 9, pages 1–87. Kluwer Academic Publishers, second edition, 2002.

[8] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

[9] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *4th Intl. Workshop on Rewriting Logic and its Applications (WRLA'04)*, *ENTCS 117*, pages 417–441. Elsevier, January 2005.

[10] N. Martí-Oliet, M. Palomino, and A. Verdejo. Strategies and simulations in a semantic framework. *Journal of Algorithms: Algorithms in Cognition, Informatics and Logic*, 62:95–116, 2007.

[11] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[12] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[13] A. Verdejo and N. Martí-Oliet. Two case studies of semantics execution in Maude: CCS and LOTOS. *Formal Methods in System Design*, 27:113–172, 2005.

[14] W3C. Web Service Description Language (WSDL) version 1.1. `http://www.w3.org/TR/wsdl`.

Digital Object Identifier: 10.4108/ICST.MOBIQUITOUS2008.3916
http://dx.doi.org/10.4108/ICST.MOBIQUITOUS2008.3916