SynchNet: A Petri Net Based Coordination Language for Distributed Objects

Reza Ziaei and Gul Agha

Department of Computer Science
University of Illinois at Urbana-Champaign, USA
{ziaei,agha}@cs.uiuc.edu

Abstract. We present SynchNet, a compositional meta-level language for coordination of distributed. Its design is based on the principle of separation of concerns, namely separation of the coordination from computational aspects. SynchNet can be used in combination with any object-based language capable of expressing sequential behavior of objects. SynchNet, which is inspired by Petri nets, has a simple syntax and semantics, but is expressive enough to code many of the commonly used coordination patterns. The level of abstraction that it provides allows tools and techniques developed for Petri nets to be readily applied to analysis and verification of the specified coordination patterns.

1 Introduction

To manage the complexity of designing distributed object systems, many proposed frameworks advocate separation of coordination from computational aspects of systems [14]. One distinct group of solutions in this category may be called the *two-level* approach. A two-level framework consists of two languages: a *base* language in which the functionality of application processes and objects is described, and a *meta* language in which the developer specifies the coordination logic that governs the interaction among application level objects. Examples of such frameworks include the Synchronizers of [19], the reflective meta-architecture of [2], and the Two-Level Actor Machine (TLAM) of [22]. The use of 'meta' vs. 'base' terminology reflects the view that meta-level coordination policies are in fact modifications to the interaction semantics of the base application.

Two-level languages usually have an involved semantics. As a result, it may be difficult to understand programs written in these frameworks and it is usually even harder to reason about them. This is especially true when the meta-level components are allowed to access the state of the base-level objects; this creates a source of interference that is difficult to control. To counter these difficulties many proposed solutions disallow meta-level coordination components to access the states of base-level objects.

Frølund [19] has proposed a coordination language and framweork in which a group of distributed objects are coordinated by entities called *synchronizers*. Each synchronizer is responsible for coordination of a group of objects: it decides

F. Pfenning and Y. Smaragdakis (Eds.): GPCE 2003, LNCS 2830, pp. 324–343, 2003.
© Springer-Verlag Berlin Heidelberg 2003

when a message *may* or *must* be delivered to an object in the group. Synchronizers do not have access to the state of coordinated objects and maintain their own independent state. The decision to approve a message delivery is based on predicates that refer to the state of the synchronizer and the information in the message. The state of the synchronizer is updated whenever an approved message is delivered. Therefore, the state of the synchronizer can be seen as an abstraction of some global snapshot of the states of the objects in the group. With this kind of abstraction, which provides a virtual local view of distributed actions, it is much simpler to solve coordination problems than with a language that only provides asynchronous message passing as a means of communication. Depending on the compiler for the synchronizer language, either centralized or distributed code may be generated.

We propose a new language called SynchNet, which follows the same design principles as Frølund's Synchronizers, but is based on Petri Nets [18]. Petri Nets is a formal modeling language for concurrent systems that has received wide academic and practical interest since its introduction by Carl Adam Petri in 1962 [18]. Its popularity is due to its rich and well-studied theory together with a friendly and easy-to-understand graphical notation. Petri Nets are less powerful than Turing machines, and therefore verification of many interesting properties is decidable [6]. Decidable properties include reachability, which is useful in verification of safety properties such as deadlock-freedom.

Using SynchNet, one can specify a synchronizer coordinating a group of objects. The specification of a synchronizer is translated into a *synchronizing net* or *synchnet*, which is in fact a Petri net. A two-level semantics relates the execution of the synchnet to method invocations in coordinated objects and thus allows enforcement of coordination requirements as specified.

The formal language of Petri Nets allows us to give formal definitions of interesting properties for synchnets. For instance, in the last section of this paper, we define a preorder relation on synchnets that states when it is safe to replace a deadlock-free synchnet with an alternative implementation while preserving the coordination properties of the first synchnet. Using this relation, one can verify the correctness of a synchnet implementation with respect to a more abstractly defined synchnet.

1.1 Related Work

Designing linguistic primitives and styles for distributed coordination, that is coordination of systems in which the primary mode of communication is asynchronous message passing, has a long history. We discuss hightlights of this evolution by first considering low-level mechanisms, and then move towards more abstract and modular constructs.

Most languages for programming communicating processes contain two operations `send` and `receive` that communicate data over *channels* connecting communicating processes. Usually a process executing a `receive` operation on a channel blocks until a message is available on the channel. Sending processes, however, may either block until a receiver is available to receive the message

(*synchronous* mode), or proceed with their execution, leaving the message in channel's *buffer* for the receiving process to pick it up later (*asynchronous* mode).

To protect blocked processes from remaining blocked whenever a channel remains empty indefinitely, the *input-guarded command* was introduced. Input-guarded command is an extenstion of Dijkstra's guarded command [5] with added conditions to check availability of messages on a channel. This construct was introduced in the language Communicating Sequential Processes by Hoare [10]. CSP uses synchronous communication, but it is not difficult to conceive of input-guarded commands in a language with asynchronous communication primitives.

A more structured and higher-level construct, which is also based on input-guarded command is Ada's *rendez-vous* mechanism [4]. Rendez-vous hides a pair of message-based communications behind an abstraction similar to a procedure call. Ada combines this procedure-like abstraction with input-guarded commands into an elegant and powerful coordination mechanism.

A practical communication abstraction, which is similar to rendez-vous, but can be virtually used with any procedural language is *Remote Procedure Call* (RPC). RPC was first introduced in the programming language Distributed Processes (DP) by Brinch Hansen [9]. RPC implementations slightly modify the procedure call semantics by translating a call/return into a pair of message communications, somewhat similar to rendez-vous. RPC is less flexible than rendez-vous as it does not allow receiving processes to choose the channel from which to receive messages. The RPC framework would require the programmer to write extensive code to avoid deadlock situtations. Yet, the simplicity and efficiency of RPC has turned it into a widely used mechanism in practice. Inspired by this success, some object-oriented languages, such as Java, extended their method invocation semantics in a similar fashion to a distributed version called *Remote Method Invocation* (RMI).

All the communication and coordination mechanisms mentioned so far suffer from a software engineering deficiency, namely the mix-up of coordination behavior with the compuational aspects of a system. To provide a separation between coordination and computation aspects, there have been many proposals for modular specification [19,3,7,16,21]. The focus of these works has mainly been on the software engineering benefits obtained from separation of concerns, such as reuse and customizability. Our proposal, while fitting in this category of work, further attempts to limit the expressivity of the language to the extent that available formal tools and theories for analysis and verification become applicable.

A useful aspect of our proposed framework is that the compiler for SynchNets automatically generates distributed code from the specification of a synchronizing net. The generated code, which is interweaved with coordinated objects' code, uses the communication primitives available in the base-language. In this sense our framework can also be placed in the more general scheme of aspect oriented programming [12], in which a separately specified aspect of a program's

behavior is automatically "weaved" into the code that implements the basic functionality of the program.

SynchNet can also be used in specifying *synchronization constraints* on the order of method invocation for a single object. It is known, however, that synchronization constraints often conflict with *inheritance* in concurrent object-oriented languages. This phenomenon is generally known as *inheritance anomaly*. Many linguistic solutions have been proposed to counter the inheritance anomaly. Matsuoka and Yonezawa provide a rather comprehensive analysis of the situation and compare various proposed solutions in [15]. They have distinguished three reasons for inheritance anomaly in this paper and show that most proposals fail to consider all. They go on by presenting a complete solution. SynchNets too, seem to successfully avoid the three sources of inheritance anomaly, despite the simplicity of their syntax and semantics.

1.2 Outline

In section 2 we motivate our approach in designing a new coordination language. In section 3 we present the syntax and semantics of SynchNet. We also present several examples to illustrate the expressive power of the language. Section 4 defines a refinement relation that states when it is safe to replace a synchronizing net with another one. Finally, we provide a summary and discuss future work.

2 Coordination of Objects with SynchNets

Our object-based model of distributed computation is inspired by the Actor model [1]. We assume each object is identified by a unique reference. Objects communicate by an asynchronous communication mechanism called ARMI (Asynchronous Remote Method Invocation). Physical locations of objects are not modeled explicitly and hence all communications are uniformly assumed to be remote. In ARMI, the source object asynchronously sends a message specifying the method of the target object to be invoked accompanied by the arguments to be passed. Messages are guaranteed to reach the target object free from error and are buffered in the target object's mailbox. No assumption is made on the order of message arrival. A local scheduler selects a message from the mailbox and invokes the specified method using the message content as arguments. Objects are single threaded and at most one method invocation can be in progress at any moment. According to this model, synchronizers are specifications that dictate the behavior of schedulers.

ARMI is similar to the remote method invocation model used in many distributed object-based languages and platforms such as CORBA [8], DCOM [20], and Java RMI [13]. The difference is that our model of invocation is asynchronous. The usual remote method invocation (RMI) is a rendez-vous like communication mechanism, in which the source object blocks until the method execution is complete and returns with a message containing the result. In ARMI, the source does not even wait for the invocation to begin. When an invoked

method reaches the end of its execution, it may choose to send back the result using a separate ARMI. Hence, it is possible to model RMI as a pair of ARMI communications. Therefore, our results can be incorporated into practical platforms that use RMI like communication mechanisms.

We propose a two-level language for coordination of distributed objects communicating via ARMI. The *base* language can be any conventional sequential class-based language such as Java or C++, with the method invocation semantics modified to be ARMI. The meta language is SynchNet. In SynchNet, coordination patterns are specified as modules. Each module is translated into a so called *synchronizing net* or *synchnet*, which is a Petri net with a slightly modified semantics that relates the transitions of the net to method invocations in the base objects. After a brief overview of Petri Nets, we motivate our work via two classical coordination problems.

2.1 Petri Nets

We begin this subsection by an informal introduction to Petri Nets and its graphical notation. A formal definition of the model will be given later. A well-written exposition on Petri Nets can be found in [17]. We also argue why Petri Nets by itself is not a suitable language for the development of distributed systems.

The graphical presentation of a Petri net is a graph with nodes and arcs connecting the nodes. There are two kinds of nodes: *places*, which usually model resources or partial state of the system, and *transitions*, which model state transition and synchronization. Arcs are directed and always connect nodes of different types. Multiple arcs between two nodes are allowed. Figure 1 is an example net.

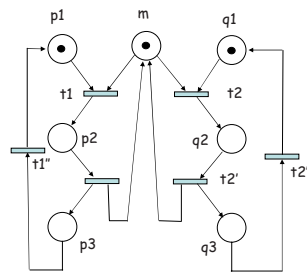


Fig. 1. A Petri Net Example

In a Petri Net, the state of the system is modeled by marking the places of the net with *tokens*. A place can be marked with a finite number (possibly zero) of tokens. For instance, in Figure 1 the token in place m represents the availability of a semaphore. The semantics of transitions determine how the state of the system changes. A transition t is said to be *enabled* in a certain marking if for every arc from a place p to t, there exists a distinct token in the marking of p. For instance, in Figure 1, the two transitions $t1$ and $t2$ are enabled. An

enabled transition can *fire* and result in a new marking. Firing of a transition t in a marking μ is an atomic operation that subtracts one token from the marking of any place p for every arc connecting p to t, and adds one token to the marking of any place p for every arc connecting t to p. For instance in Figure 1 the transition $t1$ can fire and as a result change the marking of the net by removing one token from m and one token from $p1$ and putting one token in $p2$. It is also possible for transistion $t2$ to fire. The choice is made non-deterministically.

Petri Nets is not a suitable model for distributed object-based programming. In Petri Nets asynchronous computation is represented naturally, but only synchronous communication can be modeled directly. Modeling asynchronous communication requires explicit representation of communication channels. This renders Petri Nets unfit for distributed programming. Almost every distributed programming language hides channel and buffering representations and only provides high-level primitives for communication and synchronization. Another disadvantage is that Petri Nets are not capable of directly expressing creation of new processes or objects (More expressible extensions are available but they lack the nice decidability properties of classical Petri Nets).

2.2 Example I

We state a coordination problem and write a SynchNet module to solve it. A group of transmitters are scattered in a field to transmit sensed data. Transmitters communicate with one another via asynchronous sending of messages. Delivery of messages triggers invocation of methods in the objects that control the transmitters. Each transmitter is controlled by an object with two methods: An `on` method takes an argument that determines transmission power and turns on the transmitter, and an `off` method that turns it off. A global requirement is that no two transmitters may be transmitting at the same time. It is therefore necessary that `off` messages are sent to turn off the transmitters before the next transmission begins. We abstract away the distributed logic that decides on when and to which transmitter `on` and `off` messages must be sent, and try to coordinate the global order of message delivery so that two conditions are guaranteed: (I) `on` and `off` messages are delivered to each object in alternation, (II) no two transmitters are transmitting simultaneously.

Suppose controller objects are instances of the class `TransmitterC` and that `Transmitters` is a list containing references to the identifiers of a collection of controller objects. The following module specifies the two requirements stated above:

```
synchnet TransmitterME(Transmitters: list of TransmitterC)
  init = { ob'.off | ob' in Transmitters}

  foreach ob in Transmitters [with fairness]
     method ob.on
         requires     {ob'.off | ob' in Transmitters}
```

```
         consumes     {ob.off}
     method ob.off
          requires    {ob.on}
          consumes    {ob.on}
end TransmitterME
```

To generate and install a synchnet according to the specification of `TransmittermE` on a collection of objects `G` by issuing the statement `TransmitterME(G)` in the base-language. `G` is a list of object references on which the generated synchnet must be installed.

`TransmitterME` states that an `on` method can be invoked on object `ob` if every transmitter in the group is off. In Petri net terms, it states that `ob.on` may be invoked only when in the state of `TransmitterME` there is one `ob'.off` token available for each object `ob'` in the group. Once the invocation of an `ob.on` is decided the state of the generated ynchnet is modified by adding one token corresponding to the invoked method (`ob.on` here), and consuming the tokens specified in the `consumes` multilist. Note that consuming `ob.off` here guarantees that no other `on` method is invoked unless the object `ob` is turned off again. The only requirement on invocation of an `ob.off` method is that `ob` is turned on. After consuming the token `ob.on` which indicates `ob` is on, other transmitters may get a chance to be turned on. The optional condition `[with fairness]` requires that all pending methods to objects in the group must be given a fair opportunity of invocation.

Figure 2 is the graphical version of the synchnet generated by the expression `TransmitterME({t1,t2})`, which is an instantiation of `TransmitterME` on two transmitters `t1` and `t2`. There is one place for each method of each object in the group. There is one transition for each pair of `requires-consumes` clauses specified for each method. The `requires` clause specifies tokens required for the transition to become enabled. The `consumes` clause specifies which required tokens are actually consumed and are not put back in their corresponding places. Also note that, for every transition, there is an outgoing arc to its corresponding place, and is used to record method invocation that take place.

2.3 Example II

Now we use SynchNet to solve a distributed version of the dining philosophers problem: a group of philosophers are sitting around a round table and spend their time between thinking and eating. Each philosopher needs two forks to eat but every philosopher has to share one fork with the philosopher sitting on the left and share one fork with the philosopher on the right. Philosophers may only eat if they can pick up both their forks, otherwise they have to wait for philosophers next to them to finish eating and put down the shared forks. The problem is to coordinate picking up and putting down of the forks so that every hungry philosopher gets a fair chance of eating and that a deadlock situation does not occur in which every philosopher is holding one fork while waiting for the next philosopher to release the other fork.

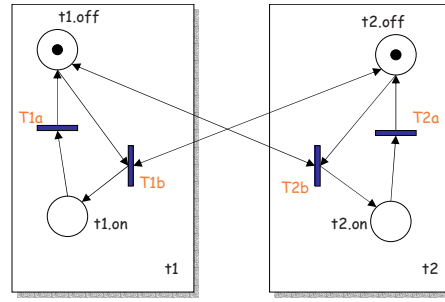


Fig. 2. Diagram of `TransmitterME` instantiated on `t1` and `t2` in its initial state

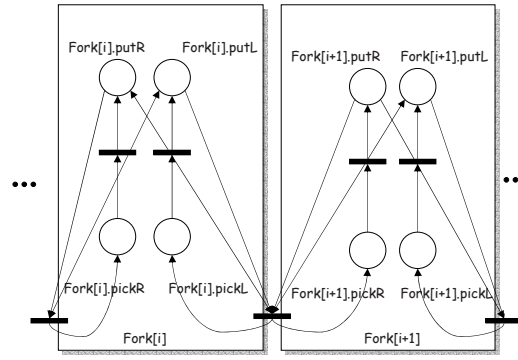


Fig. 3. Partial diagram of `Philosophers`

Suppose philosophers and forks are modeled as objects and communicate via asynchronous remote method invocation. Since we are concerned only with access to forks, we won't worry about the methods in philosopher objects. Each fork may be accessed using one of four methods: `pickL` and `putL` are used by the philosopher on the left of the fork to obtain and release the fork, and `pickR` and `putR` are used by the philosopher on the right. A local requirement is that no two back to back invocations of the form `pickX` or of the form `putX` (where `X` stands for either `L` or `R`) are allowed. There are two global requirements: fairness and deadlock-freedom. The following synchnet is a shared memory solution to this coordination problem. We assume that forks are instances of the class `Fork`. Figure 3 is a diagram of the synchnet `Philosophers`. It only depicts the part of the net corresponding to two adjacent forks. The rest of the diagram consists of similar boxes for the other forks and the whole diagram forms a cycle.

```
synchnet Philosophers(N: int, Forks : array[1..N] of Fork)
  init = {fork.putL, fork.putR | fork in set(Forks)}
```

```
  foreach fork in set(Forks) [require fairness]
     method fork.pickL
         requires     {fork.putL, fork.putR}
         consumes     {fork.putL}
     method fork.pickR
         requires     {fork.putL, fork.putR}
         consumes     {fork.putR}
     method fork.putL
         requires     {fork.pickL}
         consumes     {fork.pickL}
     method fork.putR
         requires     {fork.pickR}
         consumes     {fork.pickR}

  foreach i in {1 .. N}
     atomic(Forks[i].pickL,Forks[ (i+1) mod N].pickR)
end Philosophers
```

3 Syntax and Semantics of Synchronizing Nets

We first present the concrete syntax, followed by a translational semantics of modules into synchnets which are in fact Petri nets. Then, we present the semantics of the two-level language by relating the firing semantics of synchnets to method invocation in base-level objects. Finally, we define an inheritance mechanism that allows extension of a module with new constraints.

3.1 Syntax of Synchnets

We first present the core syntax of SynchNet. Later, we will introduce more advanced constructs as syntactic sugar. A SynchNet module has the following general form:

```
synchnet ⟨id⟩ ( ⟨param-list⟩ ) is
   init = ⟨token-list⟩
   ⟨method-clauses⟩
   ⟨atomicity-clauses⟩
end ⟨id⟩
```

where ⟨*id*⟩ is a programmer supplied identifier used to refer to the defined synchnet. ⟨*param-list*⟩ is a list of formal parameters along with their types:

$$\langle \textit{param-list} \rangle ::= \langle \textit{var}_1 \rangle \colon \langle \textit{type}_1 \rangle, \ldots, \langle \textit{var}_n \rangle \colon \langle \textit{type}_n \rangle$$

All ⟨var_i⟩'s must be distinct. ⟨$type_i$⟩s can be any type that is available in the base language, including both simple types and aggregate types such as lists or arrays. In particular, they can include class types if ⟨var_i⟩ is supposed to an

object. ⟨*param-list*⟩ acts as a binder for the specified variables, with their static scope being the body of the synchnet

The body of a module consists of a clause specifying the initial state of the corresponding synchnet and a collection of guard-transitions. Syntactically, the state of a synchnet is represented as a multilist of *token*s. Tokens correspond to the methods of the objects whose references are passed to the synchnet and are supposed to be coordinated by the synchnet. The syntax of tokens and multilists of tokens is specified as

$$\begin{aligned}
\langle \textit{token} \rangle &::= \langle \textit{var} \rangle.\langle \textit{method} \rangle \\
\langle \textit{token-list} \rangle &::= \texttt{\{}\langle \textit{token} \rangle, \ldots, \langle \textit{token} \rangle\texttt{\}} \\
&\quad | \quad \texttt{\{}\langle \textit{token} \rangle \,|\, \langle \textit{predicate} \rangle\texttt{\}} \\
&\quad | \quad \langle \textit{token-list} \rangle \texttt{ union } \langle \textit{token-list} \rangle \\
&\quad | \quad \langle \textit{token-list} \rangle \texttt{ intersect } \langle \textit{token-list} \rangle
\end{aligned}$$

where ⟨*var*⟩ must be a variable whose type is a class and ⟨*method*⟩ must be a method identifier belonging to that class. ⟨*predicate*⟩ is a predicate over object references. Predicates consists of equality or inequality constraints over variables whose types are classes, composed with the usual boolean operators. The collection of tokens inside the brackets in the expression `{` ⟨*token*⟩, ..., ⟨*token*⟩ `}` must be treated as a multiset, that is, the order of tokens is irrelevant, and the same token may appear more than once.

The body of a module consists of two kinds of coordination behavior. The first kind, ⟨*method-clauses*⟩ consists of a collection of *method clauses*. Each method clause has a header and a body. The header of the clause consists of a variable (with a class type) and a method identifier belonging to that class. The body of a method clause consists of a list of guard-transitions. A guard-transition has two parts: A *guard*, which is the condition required for the method to be invoked, and a *transition* that specifies how the state of the synchnet must change if the method is invoked. Each method clause is written as

```
method ⟨var⟩.⟨method⟩
    requires  ⟨token-list⟩
    consumes  ⟨token-list⟩
or    ...
or
    requires  ⟨token-list⟩
    consumes  ⟨token-list⟩
```

A syntactic requirement is that the consume list must be contained in the require list extended with an additional token corresponding to the method for which the require-consume pair is specified.

Atomicity is another kind of coordination requirement that can be specified in the body of a module. Atomicity requirements are represented as constraints called *atomicity clauses*. Each atomicity clause has the following format:

$$\langle \textit{atomicity-list} \rangle ::= \texttt{atomic(}\langle \textit{var}_1 \rangle.\langle \textit{method}_1 \rangle, \ldots, \langle \textit{var}_n \rangle.\langle \textit{method}_n \rangle\texttt{)}$$

where all $\langle var_i \rangle$s must refer to distinct objects.

To avoid repetitive declarations, we introduce a universal quantification operator as syntactic sugar.

```
foreach ⟨var⟩ in ⟨var-set⟩
  ⟨clauses⟩
```

where all clauses in ⟨*clauses*⟩ use the variable ⟨*var*⟩ in their headers. ⟨*var-set*⟩ is a subset of variables specified as formal arguments to the synchnet. The variable must be of class type or of aggregate types such as arrays or lists with elements being of class type. This syntactic form is equivalent to a list of clauses obtained by making copies of clauses in ⟨*clauses*⟩ each having ⟨*var*⟩ replaced with some variable in the set ⟨*var-set*⟩.

3.2 Translating SynchNet Modules to Synchnets

Let's first formalize the Petri Net model introduced in Section 2.1.

Definition 1. *Formally, a Petri net N is a four-tuple (P, T, I, O) where P is a finite set of places, T is a finite set of transitions. P and T are disjoint. $I : T \to P^\infty$ is a mapping from transitions to multisets of places. Similarly, $O : T \to P^\infty$ is a mapping from places to multisets of transitions.* □

The following is the formal definition of the graph of a Petri Net:

Definition 2. *The graph of a Petri Net $N = (P, T, I, O)$ is a bipartite directed multi-graph $G = (P \cup T, A)$ where $A = \{a_1, \ldots, a_n\}$ is a multiset of directed arcs of the form (p, t) or (t, p) for $p \in P$ and $t \in T$.* □

Now we can formalize the notions corresponding to execution of a Petri Net.

Definition 3. *A* marking *μ of a Petri Net (P, T, I, O) is a multiset of places. That is $\mu \in P^\infty$. A transition $t \in T$ is* enabled *in a marking μ if $I(t) \subseteq \mu$. An enabled transition t* fires *by subtracting $I(t)$ from μ and adding $O(t)$. That is, firing of t results in a new marking $\mu' = (\mu - I(t)) \cup O(t)$. Where $-$ and $\cup$ are taken to be multiset operations.* □

A synchnet is a Petri net and is generated when an expression of the form $S(\mathcal{O})$ is evaluated in the base language, where S is the name of a module specified in the SynchNet language and $\mathcal{O}$ is a collection of base-level object references. Now, we formally define the Petri net that constitutes the synchnet generated by evaluating the expression $S(\mathcal{O})$ given the specification of module S in SynchNet. We need to make a few assumptions before we describe the construction of a synchnet.

To avoid aliasing problems, all object references $\mathcal{O}$ passed in $S(\mathcal{O})$ must be distinct. This includes all the references contained in aggregate data structures such as arrays and lists. This sanity condition can be checked at run-time when the synchnet instance is generated. Let $\mathcal{O}$ be the collection of all object references used to create an instantiation of the module S specified in the general form below.

```
synchnet S ( V1 : T1, ..., Vn : Tn ) is
   init = I
   ...
   method Vi.Mj
      ....
   or
      requires Rijk
      consumes Cijk
   or
      ...
   ...
   AC
end S
```

For simplicity of presentation, let's assume that all $T1, \ldots, Tn$ are class types. In the general case, we ignore parameters which have a non-class type, and we expand aggregate parameters into a collection of class-type variables. The type system of the base language can be used to verify that for every pair of the form $Vi.Mj$ the method Mj actually belongs to the class Ti. We further assume that an environment $\eta : \{V1, \ldots, Vn\} \to \mathcal{O}$ is given that maps variable names to actual object references passed during the creation of the synchnet instance. For a multiset of tokens Tok, we let $\eta(Tok)$ be the multiset obtained by renaming every occurrence of $V_i.M \in Tok$ by $\eta(V_i).M$.

With these assumptions, we construct a synchnet for $S(\mathcal{O})$ in two steps. First we ignore atomicity clauses and define a Petri Net $SN = (P, T, I, O)$. If AC is non-empty, we modify SN to obtain a net $SN' = (P, T', I', O')$ that incorporates atomicity constraints specified by the list of atomicity clauses AC.

For a module S with the general form described above, let the net $SN = (P, T, I, O)$ be defined as follows. We assume that the environment η binds formal parameters (Vi) of S to object references given in $\mathcal{O}$.

- For every pair of variable V_i and method M that belongs to the class of V_i we consider a place $o.M$ with $o = \eta(V_i)$. That is $P = \{\eta(V_i).M | 1 \leq i \leq n \text{ and } M \text{ belongs to } Ti\}$.
- T is the smallest set such that for every pair of require-consume clause $(Rijk, Cijk)$ that belongs to a method $Vi.Mj$, there exists a transition $t \in T$ such that

$$I(t) = \eta(Rijk) \text{ and } O(t) = (\eta(Rijk) - \eta(Cijk)) \cup \{\eta(Vi).Mj\}$$

  If there are no require-consume pair specified for a method $Vi.Mj$ we assume there is a transition $t \in T$ such that

$$I(t) = \{\} \text{ and } O(t) = \{\eta(Vi).Mj\}$$

  we call these transitions *simple* and we say the simple transition t *corresponds to* the singleton $\{\eta(Vi).Mj\}$.

If the body of the module S contains atomicity clauses $AC_1, \ldots, AC_n$. We obtain a sequence of nets by gradually merging simple transitions into *tuple* transitions. We also keep track of merged simple transitions as the set $MT \subseteq T$, to remove them from the set of transitions after all atomic clauses are processed. Let $SN_0 = SN$ and $MT_0 = \emptyset$. For every atomicity clause AC_i $(1 \leq i \leq n)$ of the form

$$\texttt{atomic}(V_1.M_1, \ldots, V_l.M_l)$$

we modify the net $SN_j = (P, T_j, I_j, O_j)$ to obtain $SN_{j+1} = (P, T_{j+1}, I_{j+1}, O_{j+1})$ in the following way. Let T_{j+1} be the smallest set containing T_j such that for every collection of transitions $t_i \in T$ where $1 \leq i \leq l$ and t_i corresponds to $V_i.M_i$, we have a *tuple* transition $(t_1, \ldots, t_l) \in T_{j+1}$. We say that $(t_1, \ldots, t_l)$ corresponds to the set $\{\eta(V_1).M_1, \ldots, \eta(V_l).M_l\}$. We further let I_{j+1} and O_{j+1} be identical to I_j and O_j, respectively, on transitions in T_j, and for a new transition $(t_1, \ldots, t_l) \in T_{j+1}$, let $I_{j+1} = I_j(t_1) \cup \cdots \cup I_j(t_l)$ and $O_{j+1} = O_j(t_1) \cup \cdots \cup O_j(t_l)$. Finally, let $MT_{j+1} = MT_j \cup \{t_1, \ldots, t_l\}$.

By repeating the above process, we obtain $SN_n = (P, T_n, I_n, O_n)$ and MT_n. Now, let $SN' = (P, T', I', O')$ where $T' = T_n - MT_n$ and I' and O' are restrictions of I_n and O_n to T'. This completes our translation and we have SN' as the Petri net of the synchnet S.

The operational semantics of a synchronizing net $SN = (P, T, I, O)$ is a labeled transition system $(\mathcal{M}, \mathcal{L}, \mathcal{T})$ where $\mathcal{M} = P^{\infty}$ is the set of possible markings of SN, $\mathcal{L} = 2^P$ the set of labels with each label being a finite set of places, and $\mathcal{T} \subseteq \mathcal{M} \times \mathcal{L} \times \mathcal{M}$ defined as the smallest ternary relation such that if $t \in T$ is enabled in marking μ, t corresponds to the set of methods L, and μ' is the marking that results after t fires in marking μ, then $(\mu, L, \mu') \in \mathcal{T}$. We write $\mu \xrightarrow{L} \mu'$ for such a triple.

It is possible to extend SynchNet to support disjunction of atomicity constraints. Extending the synchnet construction to account for this extension is straightforward and is similar to the construction for disjuction of require-concume clauses. Due to space limitation we do not provide the construction in this paper.

3.3 Semantics of the Two-Level Language

The specification of a synchnet S is akin to a class declaration in class-based languages. To coordinate a group of objects, a synchnet must be created by the base-level program. To allow this, we extend the base language to include expressions of the form $S(Params)$. Such expressions can be added as statements if the base language is imperative or as function applications if the language is functional. The evaluation or execution of such an expression creates a new instance of S and uses *Params* to initialize and set up the Petri net corresponding to S. In general, *Params* would include references to newly created objects. A sanity check guarantees that references included in *Params* are all distinct. We require this to generate a synchnet unambiguously.

We now present the operational semantics of the two-level language. We use SI to refer to a created synchnet and assume that $\mathcal{O}$ is the set of object references coordinated by SI. Suppose μ_S denotes the state of a SI (a net marking), and σ_o the state of an object $o \in \mathcal{O}$. As stated before, we expect the base language follow the asynchronous remote method invocation protocol for communication. We don't make any assumption about the representation of the state of objects in the base language, but we assume that its formal semantics is defined as a labeled transition system with labels being either $o.\tau$ referring to some internal computation by the object o, or $o.l(v_1, \ldots, v_n)$ where o is an object reference, l is the label of some method that belongs to object references by o and $v_1, \ldots, v_n$ are actual values. The transition corresponds to the invocation of method l of object o with $v_1, \ldots, v_n$ passed as arguments. We will use the abbreviation $\tilde{V}$ for the list of values $v_1, \ldots, v_n$.

The semantics of object execution in the two-level language is defined as a labeled transition system. Suppose a synchnet SI coordinates a group of objects $\mathcal{O} = \{o_1, \ldots, o_n\}$. We let $\mathcal{S} = \{(\sigma_{o_1}, \ldots, \sigma_{o_n}, \mu)\}$, where σ_{o_i} are the local states of objects o_i $1 \leq i \leq n$ and μ is a marking of SI, be the set of global states of objects o_i coordinated by SI (we will also use the abbreviation $(\tilde{\sigma}, \mu)$). Let $\mathcal{L}_i$ be the set of transition labels that are either $o_i.\tau$ (silent or internal transition by o_i) or correspond to invocations of o_i's methods in the base language. We define a labeled transition system on global states as a triple $(\mathcal{S}, \mathcal{L}, \mathcal{T})$ where $\mathcal{L} = \mathcal{L}_1 \times \cdots \times \mathcal{L}_n$ and $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$. We use the abbreviation $\tilde{l}$ for $(l_1, \ldots, l_n)$, where $l_i \in \mathcal{L}_i$. We also write $s \xrightarrow{\tilde{l}} s'$ for $(s, \tilde{l}, s') \in \mathcal{T}$. The transition relation $\mathcal{T}$ is defined as the smallest relation satisfying the following rules

$$\frac{\sigma_{o_i} \xrightarrow{o_i.l(\tilde{V})} \sigma'_{o_i} \quad \mu \xrightarrow{o_i.l} \mu' \quad l_j = \begin{cases} o_j.\tau & \text{if } j \neq i \\ o_j.l(\tilde{V}) & \text{if } j = i \end{cases}}{(\ldots, \sigma_o, \ldots, \mu) \xrightarrow{\tilde{l}} (\ldots, \sigma'_o, \ldots, \mu')}$$

$$\frac{\forall 1 \leq i \leq n \;.\; \sigma_{o_i} \xrightarrow{o_i.l(\tilde{V})} \sigma'_{o_i} \quad \mu \xrightarrow{o_i.l} \mu'}{(\tilde{\sigma}, \mu) \xrightarrow{\tilde{l}} (\tilde{\sigma'}, \mu')}$$

$$\frac{\forall 1 \leq i \leq n \;.\; \sigma_{o_i} \xrightarrow{o_i.\tau} \sigma'_{o_i} \quad l_i = o_i.\tau}{(\tilde{\sigma}, \mu) \xrightarrow{\tilde{l}} (\tilde{\sigma'}, \mu')}$$

In words, a message $l(\tilde{V})$ sent to object o can result in invocation of $o.l$, only if the synchnet is in a state that permits the invocation. Furthermore, if the invocation takes place, the state of the synchnet changes accordingly.

3.4 Composition of Synchronizers

We can extend a synchnet specification by relaxing or further constraining the constraints specified in it. We do so via an inheritance mechanism. Suppose $S1$ is a synchnet specification, we write

```
synchnet S2 ( Params ) extends S1 is
   init = I
   ...
   method V.M
       ...
   or
       requires    [ intersect | union ] R
       consumes    [ intersect | union ] C
   or
       ...
   ...
   atomic(V1.M1,...,Vl.Ml)
   ...
end S2
```

as the specification of a synchnet $S2$ that extends the specification of $S1$. Parameters of $S1$ must be exactly the same as those of $S2$. $S2$ may refer to the initial state of its parent synchnet by the expression `Super.init`. Therefore, I can be either a multiset of tokens or the union or intersection of `Super.init` with a new multiset of tokens. The optional operators `intersect` or `union` can be used in require-consume clauses to relax or further constrain the requirements of the parent synchnet. If neither `intersect` nor `union` are specified, the multisets replace those of the parent multisets.

An independent specification for $S2$ can be obtained by a simple substitution: `Super.init` is replaced with the initial multiset of tokens defined in $S1$, and every pair of require-consume clauses of the form

```
   ...
   or
      requires    X R
      consumes    Y C
   ...
```

that belongs to method $V.M$, and where $X, Y \in \{\texttt{intersect}, \texttt{union}\}$ we replace it with

```
   ...
   or
     requires R1 X R
     consumes C1 Y C
   or
     requires R2 X R
     consumes C2 Y C
   or
     ...
```

```
or
  requires Rn X R
  consumes Cn Y C
or
  ...
```

where

```
method  V.M
  requires  R1
  consumes  C1
or
  ...
or
  requires  Rn
  consumes  Cn
```

is the complete set of require-consume clauses of $V.M$ in $S1$. The set of atomicity clauses of unwinded $S2$ is the union of atomicity clauses in $S1$ and those specified in $S2$.

3.5 More Examples

Here we present some examples to illustrate how our language may be used to modify the interactive behavior of single objects and create more familiar coordination mechanisms such as sempahores.

Example 1. In this example, we show how synchnets may be used to implement semaphores, another coordination mechanism. Suppose `ob` is some object with two methods `put` and `get`. For instance, `ob` can simply be a variable, with its content accessed via `get` invocation, and updated with invocations of `put`. The following synchnet will turn this object into a semaphore, in the sense that the number of times the `put` method is invoked always exceeds the number of times `get` is invoked. In other words, `put` will behave like the V operation of a semaphore and `get` like the P operation.

```
synchnet Sem(of : Variable)
   init = { }
   method of.put
      requires {}
      consumes {}
   method of.get
      requires {of.put}
      consumes {of.put}
end Sem
```

The guard-transition for method `of.get` indicates that every invocation of `get` requires a distinct invocation of `put` to occur in the past. As every invocation of either methods `put` or `get` adds a new token to the corresponding places,

the consume clause of `get` guarantees that the number of invocations of `get` would not exceed the number of invocations of `put`. However, because `put` does not require any tokens, `put` maybe invoked any number of times regardless of how many times `get` is invoked. This is the usual invariant requirement for semaphores, which can be intuitively verified through the simple semantics of the synchronizer language. Adding fairness to the semantics allow definitions of fair semaphores.

Example 2. A *k-bounded semaphore* has the property that at most k processes can issue a V operation that is not matched by a P operation. A 1-bounded semaphore can be defined easily by further constraining the behavior of a general semaphore.

```
syncnet  OneSem(ob : Variable) extends Sem
  init = {ob.get}
  method ob.put
     requires {ob.get}
     consumes {ob.get}
end OneSem
```

This synchronizer states that the method `put` may be invoked only if `get` had been invoked once in the past. To allow for the first `put` to go through we have modified the initial state to include a token of type `ob.get`. When this synchnet is installed on a single element variable, it turns the variable into a single-element buffer.

A *2-bounded semaphore* can be defined similarly:

```
synchnet TwoSem(ob : Variable)
   init = {ob.get,ob.get}
   method ob.put
      requires    { ob.get }
      consumes    { ob.get }
   method ob.get
      requires    { ob.put }
      consumes    { ob.put }
end TwoSem
```

Alternatively the same semaphore can be expressed using synchnet inheritance:

```
synchnet TwoSem(ob : Variable) extends OneSem
    init = {ob.get, ob.get}
end TwoSem
```

Only the initial state is modified. The rest of the synchnet specification is inherited from `OneSem`.

Example 3. Synchronizers for inherited objects may be defined compositionally using synchnet inheritance. Suppose a new class of objects `InfBuf2` is defined that adds a new method `get2` to an infinite buffer such that two elements of the buffer may be fetched at one time. The required coordination for the new class can be specified modularly and compositionally as the following synchronizer:

```
synchnet TwoBuf(A : Buffer) extends OneSem is
  init = Super.init
  method  A.get2
     requires   {A.put, A.put}
     consumes   {A.put, A.put}
end TwoBuf
```

`TwoBuf` does not modify the synchronization requirements on `put` and `get` methods and hence does not suffer from *inheritance anomaly* which many coordination mechanisms for concurrent object-oriented programming suffer [15].

4 A Preorder on Simple Synchronizers

Freedom from deadlock is an important safety property that we usually desire a collection of interacting objects to have. We define deadlock as the situation in which the state of one or more synchnet disables certain methods forever. This definition, of course, also includes the extreme case of a synchronizer disabling the invocation of all methods of an object; regardless of the behavior of the environment, this is an obvious deadlock situation.

One can verify deadlock-freedom of a synchnet by performing a reachability analysis. However, since reachability of Petri nets has non-elementary complexity, we introduce an alternative formal method for development of deadlock-free synchnets. We define a preorder relation on synchnets that allows "safe" substitution of a synchnet with an alternative implementation. In other words, we introduce a preorder relation $\leq$ over synchnet instances that is deadlock-freedom preserving: $S \leq S'$ implies that whenever S' does not deadlock in an environment E, using S' in environment E would not result in deadlock either.

The formal framework that we develop here is along the lines of the theory of failure equivalence of CSP processes presented in [11].

Definition 4. *A* trace *of a synchnet S with initial state I is a path in the labeled transition systems of the net with the root I.* □

Next we define the *failure* of a synchnet S. Intuitively, a failure describes an environment that allows S to reach a state in which all of the messages offered by the environment are blocked. A failure, therefore, consists of a pair (t, L) meaning that S can follow the trace t and end up in state μ such that none of the methods in the set L would be enabled in μ.

Definition 5. *A* failure *of a synchnet S with initial state I is a pair (t, L) where t is a trace of S starting with the marking I and ending at state μ, and L is a set of method tags such that for all $o.M \in L$, $o.M$ is not enabled in μ.*

The failures of a synchnets S with initial marking I is written as failures(S) and is the set of all failures of S starting at I. □

We are now ready to define the preorder relation on synchnets.

Definition 6. *For two synchnets S and S' that are instantiated with the same set of objects, we say $S \leq S'$ whenever failures$(S) \subseteq$ failures(S').*

We also write $S \equiv S'$ whenever $S' \leq S$ and $S \leq S'$. □

It is not difficult to see that substituting S' with S when $S \leq S'$ would not cause further deadlock situations than S would. Therefore, if synchnets are always substituted according to this preorder, a non-deadlocking synchnet would never be substituted with a one that deadlocks.

5 Discussion and Future Work

We have proposed the use of Petri Nets as a simple meta-level language to specify coordination requirements of a group of distributed objects. When this meta-level language (SynchNet) is combined with an object-based language with asynchronous remote method invocation semantics, we obtain an expressive distributed object-based language. To keep things simple, our coordination language only refers to the labels of methods. As a result, coordination requirements that discriminate between messages containing distinct values cannot be expressed in our currently proposed language. We have observed that, despite this limitation, our language is still expressive enough to represent many interesting coordination patterns.

Since synchnets are in fact Petri nets, we can benefit from the rich and well studied theory of Petri Nets. The theory includes formal characterizations of many interesting properties along with decision algorithms to decide those properties. Automatic analysis tools have made these theories accessible to practitioners.

Our compiler for SynchNet automatically generates distributed code. The current implementation uses a naive distributed shared memory protocol and therefore suffers from low performance. Currently, we are working on a new algorithm that, by exploiting the structure of synchnets, would hopefully generate distributed code with efficiency comparable to the best distributed solutions available.

Considering that most modern distributed systems operate in open and dynamic environments and that coordination requirements usually evolve throughout the lifetime of a system, it is generally desirable to have development systems that allow dynamic customization of coordination aspects. Our proposed two-level model provides some support for dynamically evolving systems: It is possible to dynamically create new objects (and threads that execute their scripts), and instantiate new synchnets to coordinate the newly created objects. More flexibility would be achieved if one could replace or modify a running synchnet on the fly. Even though our current model does not support this level of dynamic customizability, its simple and formal semantics should simplify the study of such issues.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.
2. Mark Astley, Daniel Sturman, and Gul Agha. Customizable middleware for modular distributed software. *Communications of the ACM*, 44(5):99–107, 2001.
3. Juan-Carlos Cruz and Stéphane Ducasse. Coordinating open distributed systems. In *Proceedings of International Workshop in Future Trends in Distributed Computing Systems'99*, Cape Town, South Africa, 1999.
4. U.S. Department of Defence. *Reference Manual for the Ada programming language.* DoD, Washington, D.C., January 1983.
5. E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Comm. ACM*, 8:453–457, August 1975.
6. Javier Esparza and Mogens Nielsen. Decidability issues for petri nets - a survey. *Inform. Process. Cybernet.*, 30:143–160, 1994.
7. Holger Giese. Contract-based component system design. In *HICSS*, 2000.
8. Object Management Group. *Common Object Request Broker Architecture.* OMG, 1999.
9. Per Brinch Hansen. Distributed processes: A concurrent programming concept. *Comm. ACM*, 21(11):934–941, November 1978.
10. C.A.R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):151–160, August 1978.
11. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
12. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
13. I. Kumaran and S. Kumaran. *JINI Technology: An Overview.* Prentice Hall, 2001.
14. A. Lopes, J.L. Fiadeiro, and M. Wemelinger. Architectural primitives for distribution and mobility. In *SIGSOFT 2002/FSE-10*, pages 18–22, Nov 2002.
15. Satsoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
16. Naftaly H. Minsky and Victoria Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, 2000.
17. J. Peterson. *Petri net theory and the modeling of systems.* Prentice-Hall, 1981.
18. C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, University of Bonn, Bonn, West Germany, 1962.
19. Frølund S. *Coordinating Distributed Objects: An Actor Based Approach to Coordination.* MIT Press, 1996.
20. R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects.* John Wiley and Sons, 1997.
21. Sander Tichelaar. A coordination component framework for open distributed systems. Master's thesis, University of Groningen, 1997.
22. Nalini Venkatasubramanian and Carolyn L. Talcott. Reasoning about meta level activities in open distributed systems. In *Symposium on Principles of Distributed Computing*, pages 144–152, 1995.