Mapping Constructs from AOM to CPN Tools

1 Overview of Mapping Constructs

This document aims to give a detailed description on the process of mapping AOM design models for distributed STS to CPN Tools. Once AOM design models are correctly presented in CPN Tools, they can then be formally validated and verified. In CPN Tools uses simulation analysis technique for model validation and reachability graph analysis technique for verification purposes. The following sub-sections give the overview of the mapping constructs.

1.1 Construct Structure

This subsection explains the structure of the suggested mapping constructs. The objective of these mapping constructs is to provide a step by step guidance to designers and implementers of STS for validation and verification of their design models. Since the general objective of these mapping constructs and design constructs is to help designers and developers to solve problems related to software systems, the author has decided to adopt some elements of construct structure into mapping constructs. The following are important elements that describe mapping constructs in detail.

Construct Name: This feature identifies the construct itself and gives the overview of its main functionality.

Intent: By using few sentences, this feature presents the main goal of the construct and briefly describes the solution to the given problem.

Problem Description: This feature describes the context of the problem and shows the need to solve it by answering to the question, "*why is the given construct important?*"

Solution: This feature is very important in each construct. It provides the solution to the described problem and its implementation on platform in CPN Tools for the purpose of verifying correctness of the design, simulation and visualising the behaviour of the system. Although, different designers may suggest different solutions to similar problem, it is guaranteed the given solutions are correct. The specific implementations of given constructs follows in Section 1.2.

1.2 A List of Mapping Constructs

According to recent literature, researchers have shown a strong interest to consider human factors during software development life style. Among these researchers, Sterling and Taveter, suggested an approach named Agent Oriented Modelling (AOM), for engineering such systems from agent-oriented perspective. For many years, AOM has been focusing on the analysis and design of STS without proper mechanism of validation and verification. The following table summarises the mapping constructs from AOM to CPN Tools for verification and validation purposes.

Table 1: Summary of mapping constructs from AOM to CPN Tools

ID	**Construct Name**	**Intent**
1	Incoming Message	To describe a message transferred by an agent instance to another agent instance.
2	Outgoing Message	To describe a response message by an agent after successfully receiving the message from another agent.
3	Non-Communicative Action Event	To describe a physical event by a human agent or a hardware device during the interaction

4	Agent Initialisation	To identify and show the availability of an agent instance in an open and distributed socio-technical system.
5	Composite Activity	To describe the behaviour of an agent that needs to perform an activity composing a set of sub-activities.
6	Reactive Event	To describe the behaviour of an agent after perceiving changes in the environment.
7	Looping Condition	To allow an agent execute the same activity repeatedly as far as a given pre-condition(s) or post-condition(s) holds.
8	Conditional Activity	To prevent an agent to execute a particular activity until fulfilling a given condition.
9	Parameters Passing Between Activities	To allow an agent to transfer knowledge from one activity to another activity.
10	Receive Message	To describe the behaviour of an agent receiving a message sent by another agent asynchronously.
11	Send Message	To describe the behaviour of an agent sending asynchronous message to another agent.
12	Perceive Non-Communicative Action Event	To describe the behaviour of a software agent perceiving non-communicative action event from human or hardware device.

2 Catalogue of Mapping Constructs

This section presents the mapping constructs and categorises them into three main viewpoint aspects in Agent Oriented Software Engineering, namely knowledge, interaction and behaviour. The following subsection explains mapping constructs related to knowledge aspect.

2.1 Knowledge Constructs

Agents need sufficient amount of knowledge in order to execute its behaviour, especially make decisions. The foundation of these knowledge constructs comes from abstract role-based domain model and more concreate knowledge model presented in Section 3.4.1 and Section 3.4.2 respectively. An agent knowledge can either be about itself, other agents or objects in its environment. An agent uses knowledge attributes to describe itself and uses conceptual objects to describe knowledge about other agents and objects in its environment. The following constructs describe the means to map knowledge attributes and conceptual objects to CPN tools in order to simulate and verify platform independent design models of STS.

Construct Name: Knowledge Attributes

Intent: To represent one or more quality dimension of an agent

Problem Description: Agent uses knowledge attributes to represent quality dimension about itself. For example, each person can be characterised by the date of birth, height, weight, hair colour, eye colour, and so forth. This way, an agent can easily differentiate itself from other agents in STS, hence easily identified. The heuristics for knowledge modelling in Section 3.4.2 describes the most frequent data types for knowledge attributes are String, Integer, Real, Boolean, Date and Enumeration. The challenge is to identify and design a better way to represent knowledge attributes in CPN Tools.

Solution: ….

Construct Name: Conceptual Objects

Intent: To represent knowledge of an agent about other agents and objects in its environment.

Problem Description: Agent uses conceptual objects to represent knowledge about other agents and objects in its environments. The latter includes resources consumed by agents. The description of conceptual objects normally uses knowledge attributes. For example, a medical prescription can be characterised by patient name, patient address, prescriber name, prescriber address, prescriber registration number, drug(s), and so forth. This way, an agent can therefore possess or share a large amount of information using only one conceptual object. The challenge is to identify and design a suitable manner to represent conceptual objects in CPN Tools.

Solution: ….

2.2 Interaction Constructs

Construct Name: Incoming Message

Intent: To describe a message transferred by an agent instance to another agent instance.

Problem Description: Distributed agents are located in different places, either physically or virtually. Additionally, the agents exchange messages through interactions for the purpose of achieving common goals. Therefore, it is becomes important to ensure a sent message is delivered by intended agent instance.

For example, Figure 2.1 shows interaction sequence diagram between two agent instances of type Sender and Receiver. The exchanged knowledge is contained in an instance of MessageType1 which is an incoming message with respect to agent instance Receiver.

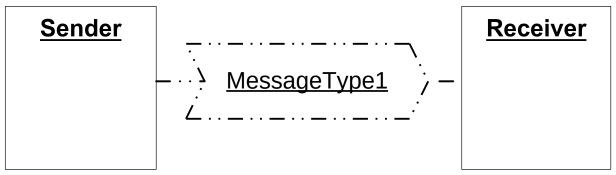


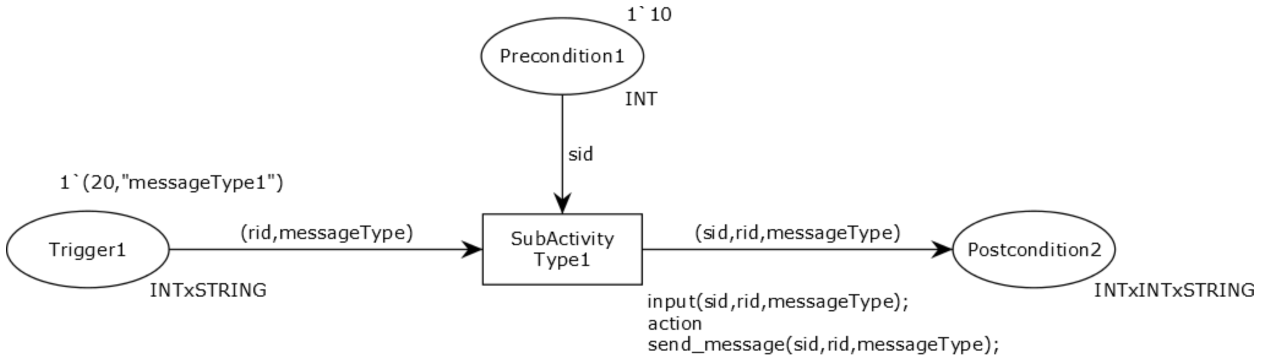
Figure 2.1: An agent sending a message

Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	SubActivityType1	Trigger1	Precondition1	Postcondition1 Postcondition2
2	SubActivityType2	Trigger2	Precondition2	Postcondition3

Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val sender = "Sender";
     val receiver = "Receiver";
     val _ = MSC.addProcess(msc,sender);
     val _ = MSC.addProcess(msc,receiver);

colset
     colset INT = int;
     colset STRING = string;
     colset INTxSTRING = product INT*STRING;
     colset INTxINTxSTRING = product INT*INT*STRING;

variables
     var messageType: STRING;
     var sid, rid: INT;

functions
     fun send_message(senderID, receiverID, message)=
     MSC.addEvent(msc,sender,receiver,"SUB-ACTIVITY 1:"^message);
```



Construct Name: Outgoing Message.

Intent: To describe a response message by an agent after successfully receiving the message from another agent.

Problem Description: Through interaction process an agent may receive more than one message from a single or many agents. It is therefore very important to clearly describe response message for each received message and ensure delivery of the message by an appropriate (intended) agent.

According to the Fundamentals of Intelligent Physical Agents (FIPA) [72], an agent needs to send an acknowledgement to the appropriate agent(s) confirming the receipt of a given message. Figure 2.2 shows an interaction diagram depicting acknowledgement instance MessageType1 sent by the receiver agent to sender agent.

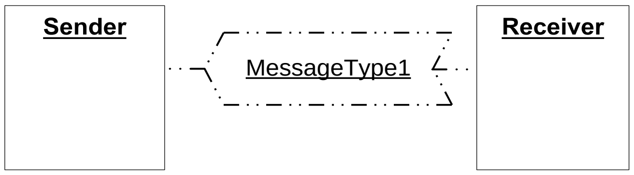


Figure 2.2: An agent receiving a message

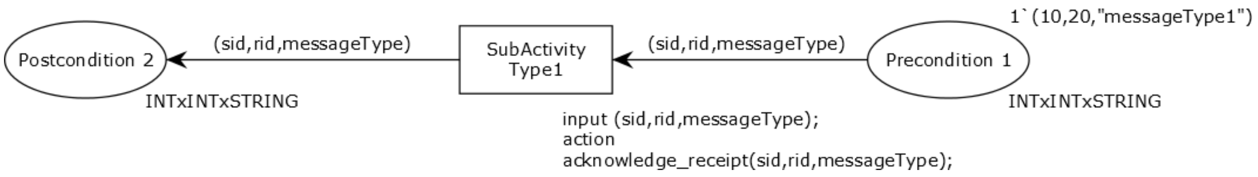
Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	ActivityType1	Trigger 1		Postcondition1 Postcondition2
2	ActivityType2	Trigger 2	Precondition 2	Postcondition 3

Trigger1 → Precondition1
Trigger2 → Postcondition2

Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val sender = "Sender";
     val receiver = "Receiver";
     val _ = MSC.addProcess(msc,sender);
     val _ = MSC.addProcess(msc,receiver);

colset
     colset INT = int;
     colset STRING = string;
     colset INTxINTxSTRING = product INT*INT*STRING;

variables
     var messageType: STRING;
     var sid, rid: INT;

functions
     fun acknowledge_receipt(senderID, recieverID,message)
     =MSC.addEvent(msc,receiver,sender,"SUB-ACTIVITY
     1:"^message);
```



Construct Name: Non-Communicative Action Event

Intent: To describe a physical event by a human agent or a hardware device during the interaction

Problem Description: Among the main characteristics of sociotechnical systems is consideration of humans, software and hardware devices during the design phase [1]. When designing interaction diagrams, it becomes crucial to differentiate communicative action event (message) performed by a software from physical action event commonly referred as non-communicative action event performed by humans or hardware devices.

Figure 2.3 describes non-communicative action instance ActionType1 performed by the sender agent on the receiver agent. The sender performing non-communicative action can either be a human or hardware device. This construct is similar to the construct for sending a message. The only difference is in the notation that describes an instance of action type. In

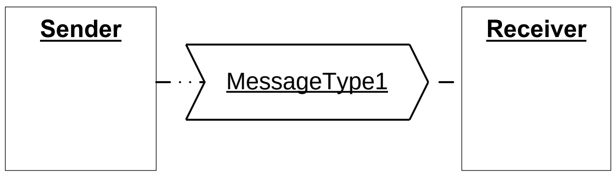


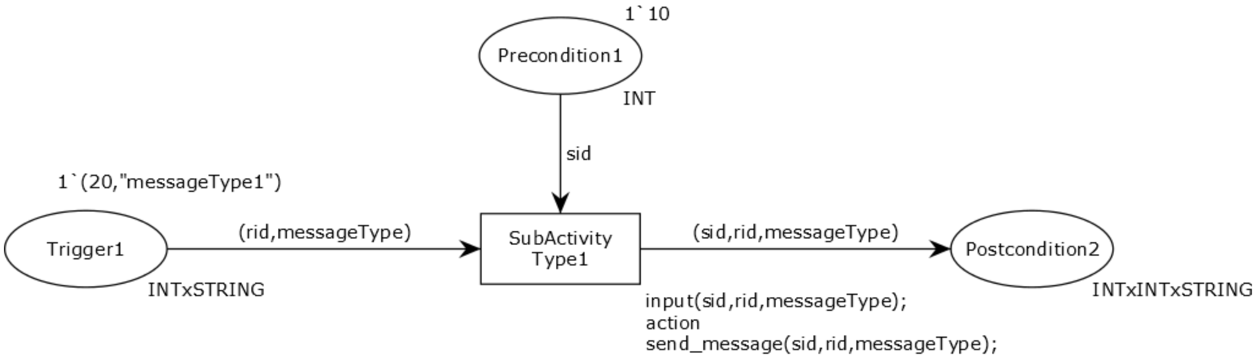
Figure 2.3: Non-communicative action event by a human or hardware device

Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	SubActivityType1	Trigger1	Precondition1	Postcondition1
				Postcondition2
2	SubActivityType2	Trigger2	Precondition2	Postcondition3

Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val sender = "Sender";
     val receiver = "Receiver";
     val _ = MSC.addProcess(msc,sender);
     val _ = MSC.addProcess(msc,receiver);

colset
     colset INT = int;
     colset STRING = string;
     colset INTxSTRING = product INT*STRING;
     colset INTxINTxSTRING = product INT*INT*STRING;

variables
     var messageType: STRING;
     var sid, rid: INT;

functions
     fun send_message(senderID, receiverID, message)=
     MSC.addEvent(msc,sender,receiver,"SUB-ACTIVITY 1:"^message);
```



2.3 Behaviour Constructs

Construct Name: Agent Initialization

Intent: To identify and show the availability of an agent instance in an open and distributed socio-technical system.

Problem Description: In an open and distributed sociotechnical system, each collaborating agent type is represented by one or many agent instances that enter and leaves the system at any time. Therefore, during initialisation process of an agent, it becomes important to register each instance of any agent type.

In the registration activity, an agent instance acquires unique identifier and makes itself ready for collaboration, i.e., receiving or sending of action events. For example, in Java Agent Development (JADE) framework, Agent Management System (AMS) service is responsible to register agents when enter the system and deregister when leave the system [73]. Figure 2.4 describes initialisation process of an instance of AgentType1.

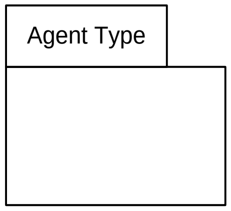


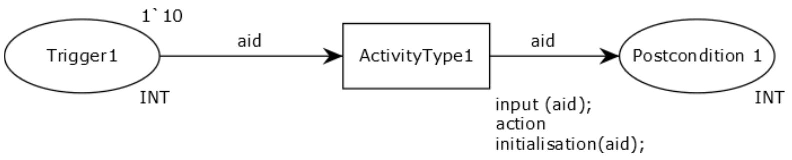
Figure 2.4: Agent Initialisation

Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	ActivityType1	Trigger 1		Postcondition1

Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val agent = "Agent";
     val _ = MSC.addProcess(msc,agent);

colset
     colset INT = int;

variables
     var aid: INT;

functions
     fun initialisation(agentID)=
     MSC.addInternalEvent(msc,agent,"INITIALIZE:"^INT.mkstr(agent
     ID));
```



Construct Name: Composite Activity

Intent: To describe the behaviour of an agent that executes an activity containing a set of sub-activities.

Problem Description: It is common to find an activity composed of a set of sub-activities. The execution of a set of sub-activities gives the output to the main activity. It is therefore important to correctly describe the connection between the main activity and the composed set of sub-activities.

For the main activity and each sub-activity, there is input and output. Since sub-activities are contained in the main activity, the input of the first sub-activity must be the same as the input of main activity. Similarly, the output of the main activity must be the same to the out-put of the last sub-activities. Figure 2.5 illustrates an instance of the main activity containing two instances of sub-activities that are executed in a sequential manner.

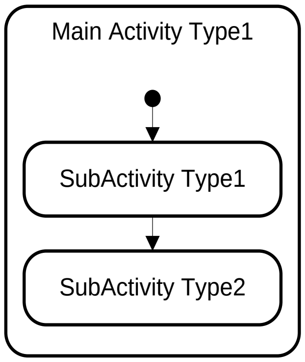
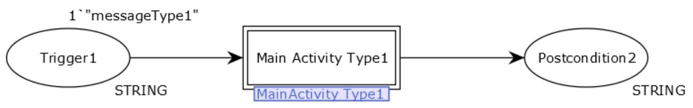


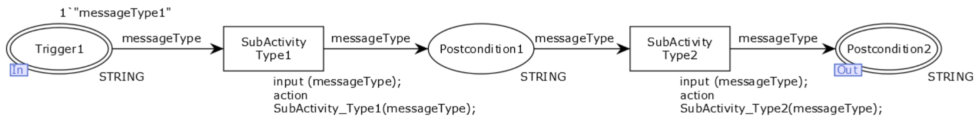
Figure 2.5: A composite activity containing two sub-activities.

Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	MainActivity Type1	Trigger 1		Postcondition1

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	SubActivity Type1	Trigger 1		Postcondition2
2	SubActivity Type2	Trigger 2		Postcondition1





Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val agent = "Agent";
     val _ = MSC.addProcess(msc,agent);

colset
     colset STRING = string;

variables
     var messageType: STRING;

functions
     fun SubActivity_Type1(message)=
          MSC.addInternalEvent(msc,agent,"CONTENT:"^message);
     fun SubActivity_Type2(message)=
          MSC.addInternalEvent(msc,agent,"CONTENT:"^message);
```

Construct Name: Reactive Behaviour

Intent: To describe the behaviour of an agent after perceiving changes in the internal or external environment.

Problem Statement: Reactivity is among important characteristics of an agent in socio-technical systems and multi-agent systems in general. Reactivity is a system behavior in which every single agent in the system copes with the environmental changes by providing a specific solution to reorganize its own task in order to fulfill the accomplishment of its originally assigned goal [74]. A reactive agent continuously observes the environment and detects changes that trigger certain behaviours after satisfying given conditions. Figure 2.6 describes a reactive behaviour of an agent that is triggered when a condition contained in rule R1 is fulfilled by changes in the environment. A given agent executes Sub-Activity Type1 in response to the environmental change that fulfils the condition. Otherwise the agent executes Sub-Activity Type2.

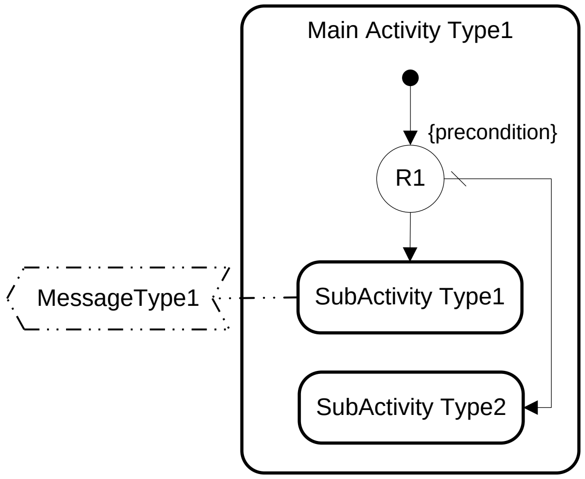


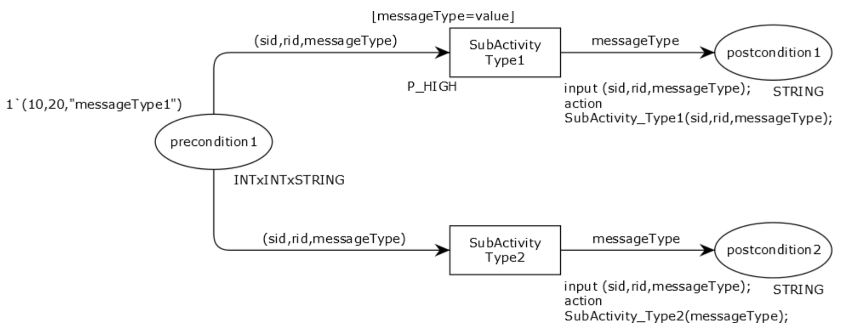
Figure 2.6: Reactive behaviour of an agent

Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	SubActivity Type1		Precondition1	Postcondition1
2	SubActivity Type2		Precondition1	Postcondition2

Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val sender = "Sender";
     val receiver = "Receiver";
     val _ = MSC.addProcess(msc,sender);
     val _ = MSC.addProcess(msc,receiver);

colset
     colset INT = int;
     colset STRING = string;
     colset INTxINTxSTRING = product INT*INT*STRING;

variables
     var messageType: STRING;

values
     val value = “messageType1”;


functions
     fun SubActivity_Type1(senderID, receiverID, message)=
     MSC.addEvent(msc,sender,receiver,"CONTENT:"^message);

     fun SubActivity_Type2(message)=
     MSC.addInternalEvent(msc,sender,"CONTENT:"^message);
```



Construct Name: Looping Condition

Intent: To allow an agent to execute the same activity repeatedly as far as a given pre-condition(s) or post-condition(s) holds.

Motivation: Agent behaviour consists of a set of activities, each activity contains at least one pre-condition and one post-condition. When executing its activities, sometimes an agent needs to execute the same activity repeatedly as far as a given pre-condition or post condition holds.

Figure 2.7(a) describes pre-conditional looping behaviour that occurs when an agent executes Sub-Activity Type 1 repeatedly as far as the pre-condition contained in rule R1 holds, otherwise executes Sub-Activity Type 2. In the other hand, Figure 2.7(b) describes post-conditional looping behaviour that occurs when an agent executes Sub-Activity Type 1 repeatedly as far as the post-condition contained in rule R1 holds, otherwise executes Sub-Activity Type 2. The main difference between these two conditional looping is that, in pre-condition looping the minimum number of executing Sub-Activity Type 1 is zero while in post-condition looping, the minimum of executing Sub-Activity Type 1 is one.

Problem Description:

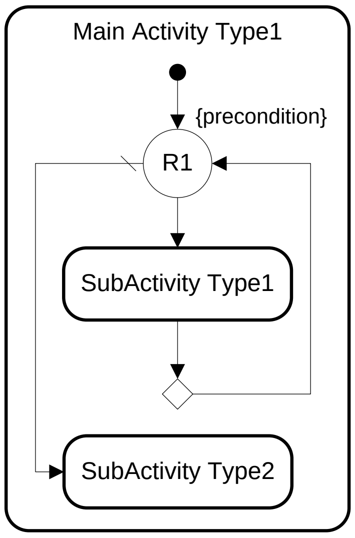


Figure 2.7(a): Pre-condition looping

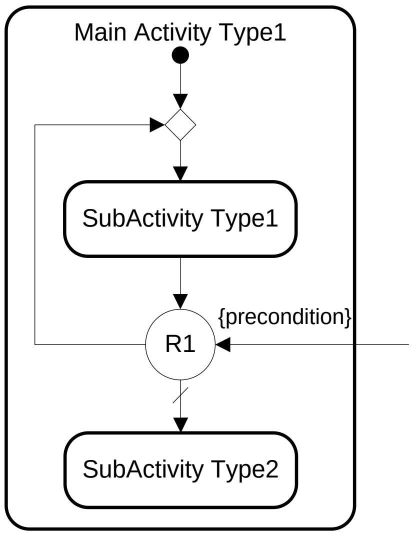


Figure 2.7(b): Post-condition looping

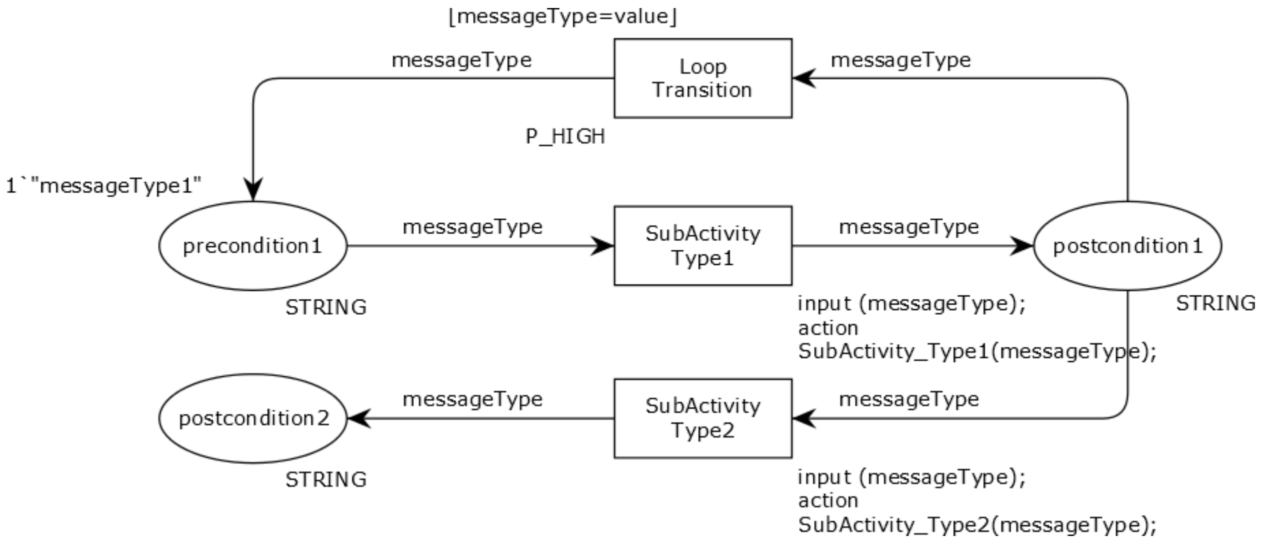
Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	SubActivity Type1		Precondition1	Postcondition1
2	SubActivity Type2		Precondition1	Postcondition2

Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val sender = "Sender";
     val _ = MSC.addProcess(msc,sender);

colset
     colset STRING = string;
```

```
variables
     var messageType: STRING;

values
     val value = “messageType”;


functions
     fun SubActivity_Type1(message)=
     MSC.addInternalEvent(msc,sender,"SUB-ACTIVITY 1:"^message);
     fun SubActivity_Type2(message)=
     MSC.addInternalEvent(msc,sender,"SUB-ACTIVITY 2:"^message);
```

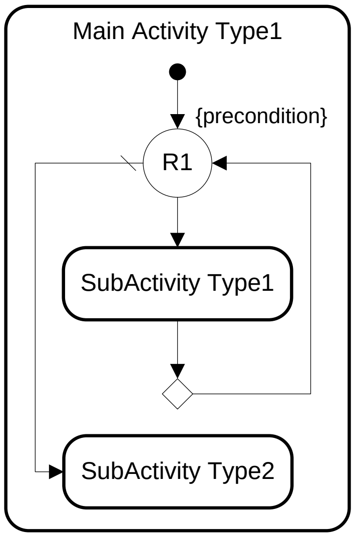


Construct Name: Looping Condition

Intent:

Motivation:

Problem Description:



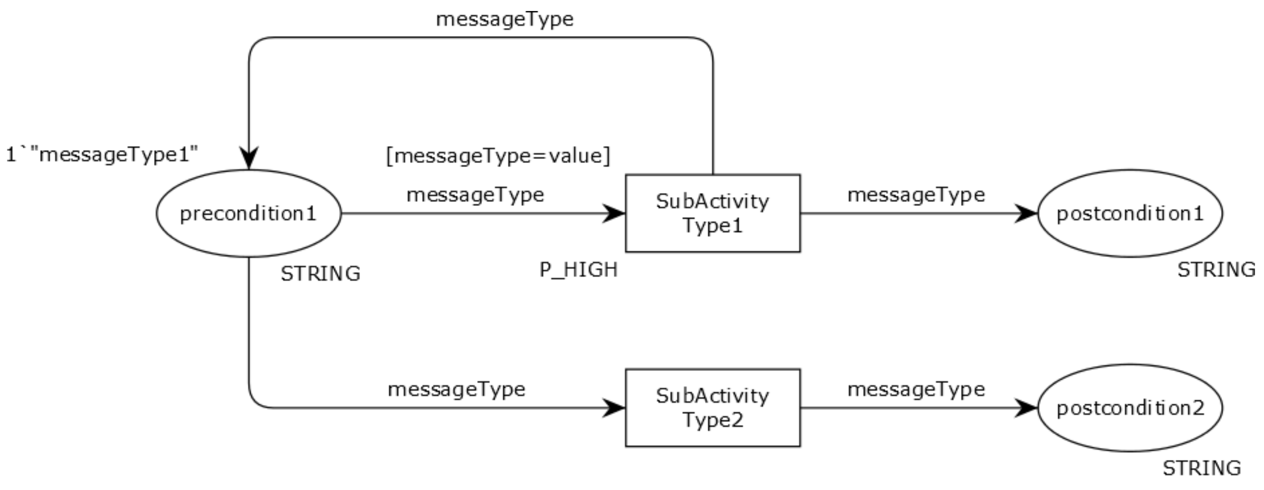
Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	SubActivity Type1		Precondition1	Postcondition1
2	SubActivity Type2		Precondition1	Postcondition2

Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val sender = "Sender";
     val _ = MSC.addProcess(msc,sender);

colset
     colset STRING = string;

variables
```

```
     var messageType: STRING;

values
     val value = "messageType";


functions
     fun SubActivity_Type1(message)=
     MSC.addInternalEvent(msc,sender,"SUB-ACTIVITY 1:"^message);
     fun SubActivity_Type2(message)=
     MSC.addInternalEvent(msc,sender,"SUB-ACTIVITY 2:"^message);
```



Construct Name: Rule-Based Activity

Intent: To allow an agent to reason in order to execute the most appropriate activity.

Problem Description: All agents in socio-technical system aim to effectively and efficiently achieve the main purpose of the system [75], i.e., main goal of the system. Since the multi-agents environment keeps on changing, often each agent needs to reason and decide appropriate set of activities to execute in order to achieve the intended goal [76]. Reasoning is among the main characteristics of an agent that is achieved through execution of rules stored in rule engine. Figure 2.8 depicts a rule-based activity where an agent executes Sub-Activity Type 1 when the condition contained in rule R1 is fulfilled.

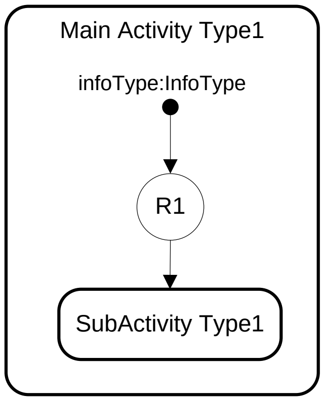


Figure 2.8: Rule-based activity

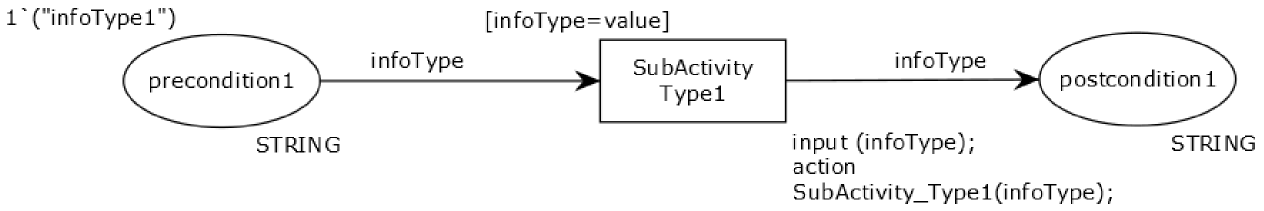
Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	SubActivity Type1		Precondition1	Postcondition1

Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val sender = "Sender";
     val _ = MSC.addProcess(msc,sender);

colset
     colset STRING = string;

variables
     var infoType: STRING;

values
     val value = “infoType1”;
```

```
functions
     fun SubActivity_Type1(info)=
     MSC.addInternalEvent(msc,sender,"SUB-ACTIVITY 1:"^info);
```



Construct Name: Parameter Passing Between Activities

Intent: To allow an agent to transfer knowledge from one activity to another activity.

Problem Description: Normally an agent performs a set of activities when executing certain behaviour. In many cases, these activities depend on each other, i.e., the output of a given activity becomes the input of the following activity. Therefore, it is important to enable an agent to seamlessly transfer knowledge between two successive activities. Figure 2.9 describes an agent transferring knowledge from Activity Type1 to Activity Type 2.

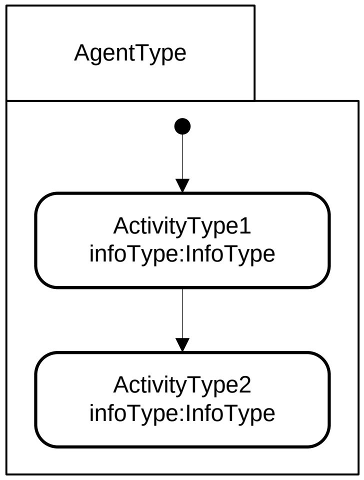


Figure 2.9: An agent transfer knowledge between activities

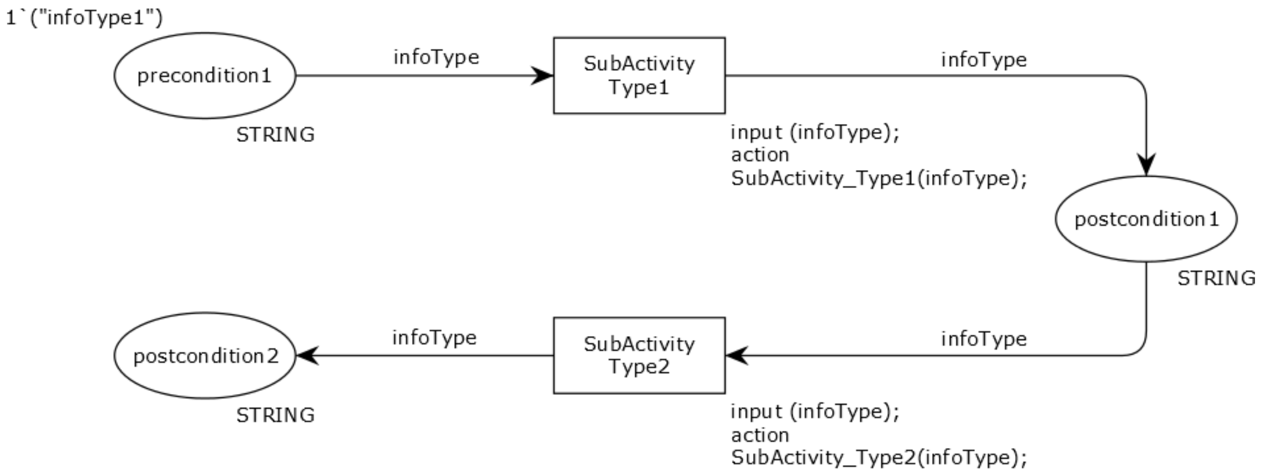
Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	SubActivity Type1		Precondition1	Postcondition1
2	SubActivity Type2		Precondition1	Postcondition2

Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val sender = "Sender";
     val _ = MSC.addProcess(msc,sender);

colset
     colset STRING = string;

variables
```

```
     var infoType: STRING;



functions
     fun SubActivity_Type1(info)=
     MSC.addInternalEvent(msc,sender,"SUB-ACTIVITY 1:"^info);
     fun SubActivity_Type2(info)=
     MSC.addInternalEvent(msc,sender,"SUB-ACTIVITY 2:"^info);
```



Construct Name: Receiving Message

Intent: To describe the behaviour of an agent receiving asynchronous message sent by another agent.

Motivation: One of the key concepts of multi-agent system is asynchronous communication, where the sender transmits data to the receiver, generally without the use of an external clock signal [77]. In other words, the sender may perform communicative action event while the receiver is offline.

Each agent has its own goal(s) and behaviours that execute a set of activities. Additionally, an agent may enter and leave the system at any time. For this reason, it becomes important to identify necessary condition(s) that may trigger an activity for receiving asynchronously sent message. Figure 2.10 describes agent behaviour for receiving asynchronous message. When the necessary conditions contained in rule R1 are fulfilled, an agent receives the instance of MessageType1 and performs Sub-Activity Type1.

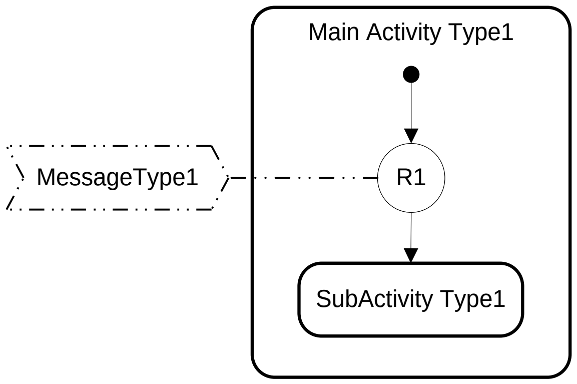


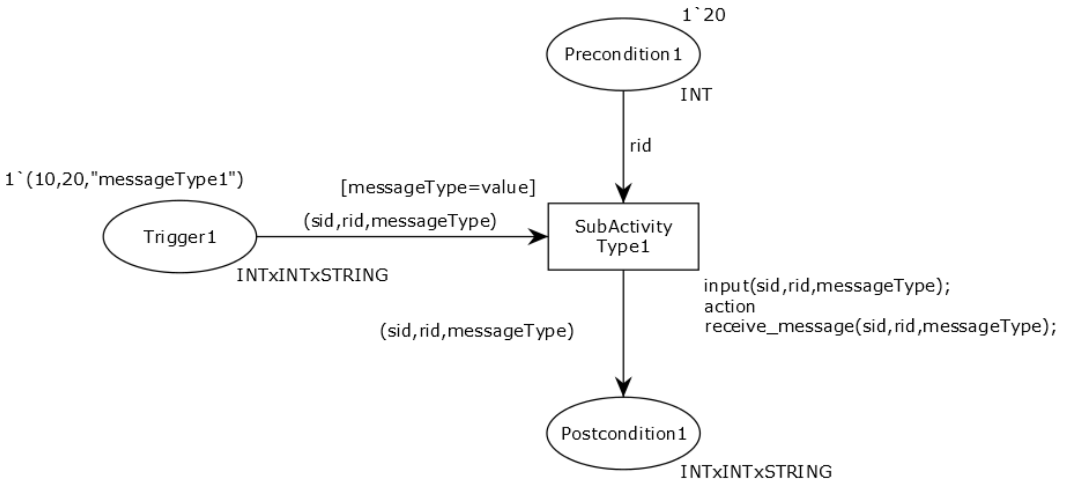
Figure 2.10: An agent receiving a message

Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	SubActivity Type1	Trigger1	Precondition1	Postcondition1

Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val sender = "Sender";
     val receiver = "Receiver";
     val _ = MSC.addProcess(msc,sender);
     val _ = MSC.addProcess(msc,receiver);

colset
     colset INT = int;
     colset STRING = string;
     colset INTxINTxSTRING = product INT*INT*STRING;

variables
     var messageType: STRING;
     var sid, rid: INT;

values
     val value = "messageType1";


functions
     fun receive_message(senderID,receiverID, message)=
     MSC.addEvent(msc,sender,receiver,"SUB-ACTIVITY 1:"^message);
```



Construct Name: Sending Message

Intent: To describe the behaviour of an agent sending asynchronous message to another agent.

Problem Description: Agents in socio-technical systems that are naturally distributed work in collaboration and pursue assigned tasks to achieve the overall goal of the system [78]. Agents often need to receive help other agents and therefore send messages to request services offered by other agents.

When designing agent behaviour it is important to specify necessary condition(s) that may trigger an activity for sending message. Figure 2.11 describes the behaviour of agent sending a message. Sub-Activity Type1 for sending instance of Message Type1 executes when an agent fulfils the condition stated in rule R1.

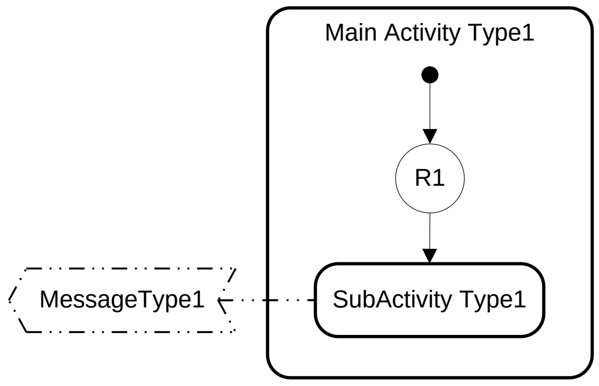


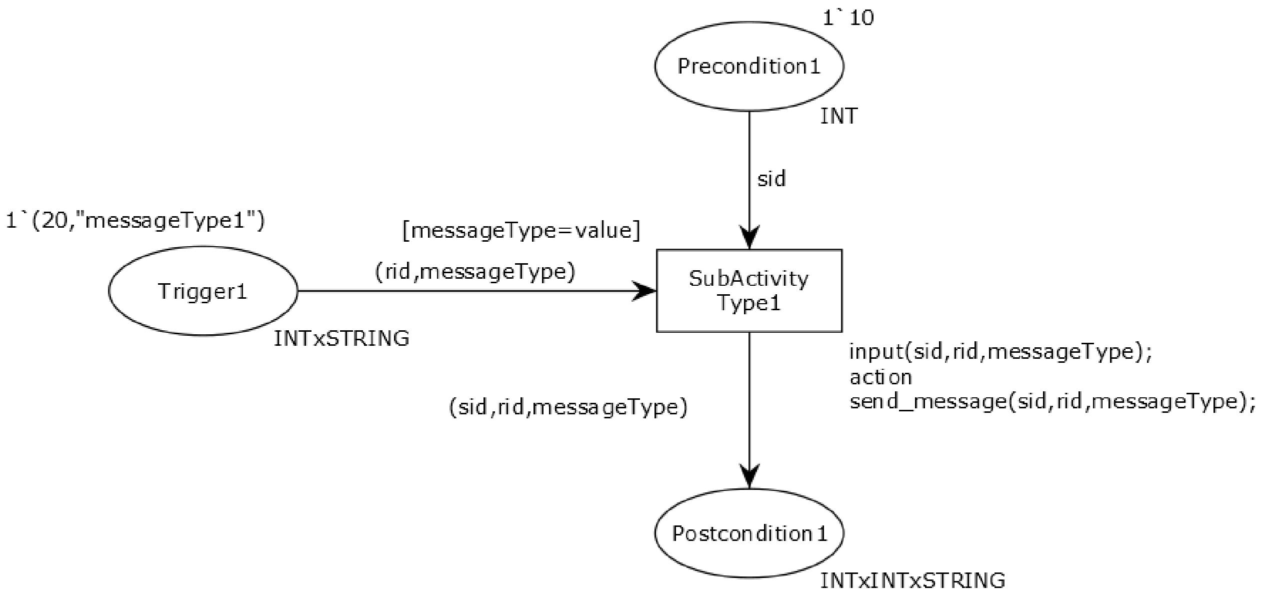
Figure 2.11: An agent sending a message

Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	SubActivity Type1	Trigger1	Precondition1	Postcondition1

Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val sender = "Sender";
     val receiver = "Receiver";
     val _ = MSC.addProcess(msc,sender);
     val _ = MSC.addProcess(msc,receiver);

colset
     colset INT = int;
     colset STRING = string;
     colset INTxSTRING = product INT*STRING;
     colset INTxINTxSTRING = product INT*INT*STRING;

variables
     var messageType: STRING;
     var sid, rid: INT;

values
     val value = "messageType1";


functions
     fun send_message(senderID,receiverID, message)=
     MSC.addEvent(msc,sender,receiver,"SUB-ACTIVITY 1:"^message);
```



Construct Name: Perceiving Non-Communicative Action Event

Intent: To describe the behaviour of a software agent perceiving non-communicative action event from human or hardware device.

Problem Statement: Non-communicative action event occurs when the event, usually physical event is initiated by human agent or hardware device [79]. In contrast to receiving message construct, software agents perceive non-communicative action events take into consideration physical actions by other agents. It is then important to clearly describe the behaviour of an agent perceiving non-communicative action event.

Figure 2.12 describes agent behaviour for perceiving an instance of non-communicative action event ActionType1. When the necessary conditions contained in rule R1 are fulfilled, an input management activity starts. Input management activity consists of rule R2 that contains condition(s) necessary for handling input action events.

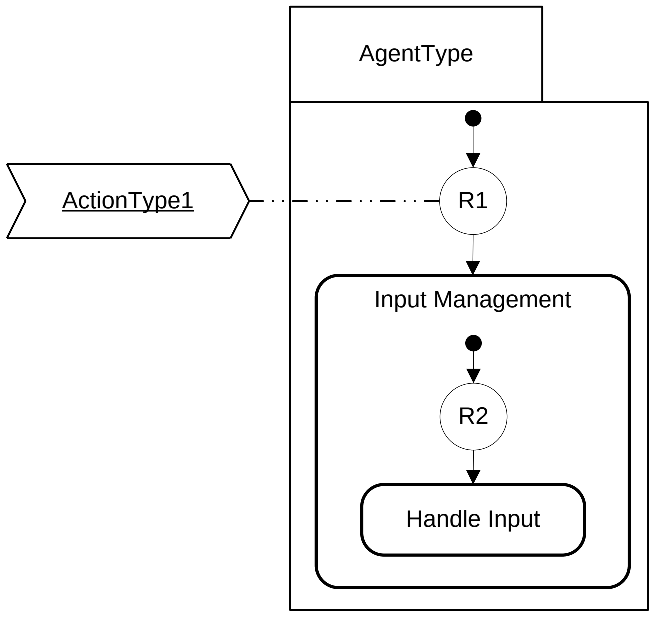


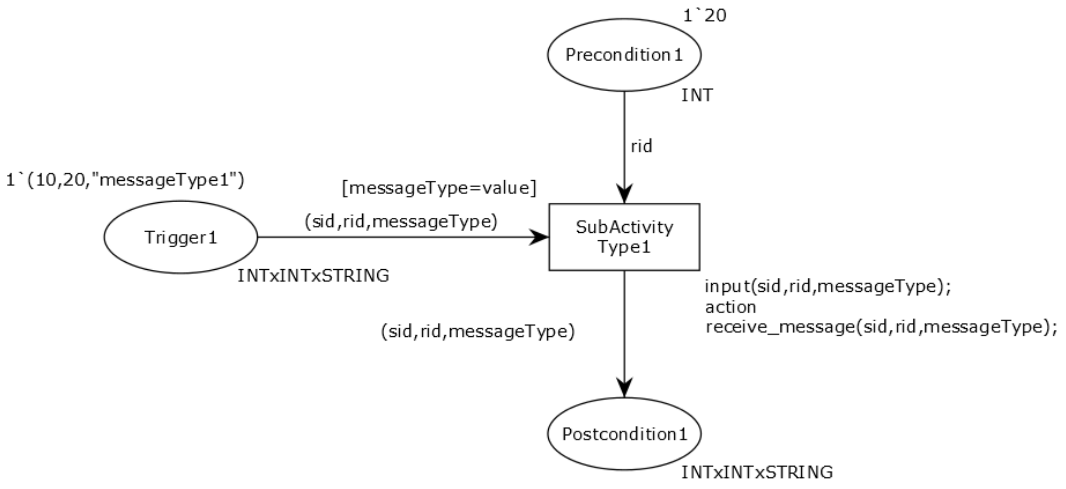
Figure 2.12: An agent perceiving non-communicative action event

Solution:

AID	Activity Name	Trigger	Precondition(s)	Postcondition(s)
1	SubActivity Type1	Trigger1	Precondition1	Postcondition1

Standard declarations

```
MSC Setup
     val msc = MSC.createMSC("Sequence Diagram");
     val sender = "Sender";
     val receiver = "Receiver";
     val _ = MSC.addProcess(msc,sender);
     val _ = MSC.addProcess(msc,receiver);

colset
     colset INT = int;
     colset STRING = string;
     colset INTxINTxSTRING = product INT*INT*STRING;

variables
     var messageType: STRING;
     var sid, rid: INT;

values
     val value = “messageType1”;


functions
     fun receive_message(senderID,receiverID, message)=
     MSC.addEvent(msc,sender,receiver,"SUB-ACTIVITY 1:"^message);
```


1`20
Precondition1
INT
rid
1`(10,20,"messageType1")
[messageType=value]
Trigger1
(sid,rid,messageType)
SubActivity Type1
INTxINTxSTRING
input(sid,rid,messageType);
action
receive_message(sid,rid,messageType);
(sid,rid,messageType)
Postcondition1
INTxINTxSTRING