SOCCA Basics

P.J. 't Hoen[*] L.P.J. Groenewegen[†] V. Zweije[‡]

November 18, 1999

Abstract

This paper gives a brief introduction to SOCCA (Specification of Coordinated and Cooperative Activities).

Contents

[*]hoen@wi.LeidenUniv.nl
[†]luuk@wi.leidenuniv.nl
[‡]zweije@wi.leidenuniv.nl

1 Introduction

This paper introduces SOCCA (Specification of Coordinated and Cooperative Activities) as presented in [EG94] and formalised in [DGSK$^+$99]. We have repeatedly experienced that the basic SOCCA concepts are difficult to understand at first reading. This paper is a simple introduction to the SOCCA concepts. This paper is different from other SOCCA publications as it does not claim to be complete, innovative, formal or to cover the extensions of SOCCA. We do not for example discuss the advanced applications of the SOCCA concepts. It is however intended to be short and easy to understand and to serve as a starting point for first reading before more complex material is tackled such as [DGSK$^+$99] or [tHDG$^+$99].

SOCCA as originally developed is strongly based upon OMT [RBP$^+$91]. OMT has been integrated in its successor, UML [UML99]. SOCCA has followed this development and now strongly leans on UML for the notation of the model elements. The strength and innovation of SOCCA is the precise modelling of the communication and coordination between the classes/ objects of the model. This paper focuses exclusively on the concepts required to understand how the communication and coordination between classes/objects is defined in SOCCA.

2 SOCCA

The four perspectives in SOCCA that are currently mature and formalised in [DGSK$^+$99] are:

The data perspective which focuses on the static, structural aspects of models.

The behaviour perspective which focuses on dynamic aspects of individual classes and objects which are made available to other classes and objects.

The functionality perspective which focuses on dynamic aspects of individual classes and objects that are internal to them.

The communication perspective which focuses on the communication between individual classes and objects.

We explain the basic concepts of SOCCA perspective by perspective and in the order listed. The communication perspective easily presents the most problems for first time readers. The next section begins with a tour of the data perspective. Throughout the text we assume that the reader has at least a rudimentary understanding of object-oriented concepts.

2.1 Data Perspective

The *data perspective* in a SOCCA model describes the static structural aspects of the model. This perspective is represented by a class diagram. The data perspective describes the classes and the relationships. The inheritance, aggregation and general association relationships are drawn in this class diagram.

Special for SOCCA is the *uses* relationship between classes. In SOCCA, there is no implicit import within a class; methods within a class are *not* automatically available for use by other methods within the class. Thus, even within a class or an object, one method can only call another method if it has a uses relationship to that class with that method in the labelling set.

The reason for making this import within a class explicit is that this import has consequences for other parts of a SOCCA model: namely, those parts that deal with coordination of communication by method calls.

For SOCCA, method invocations within one object can in principle be executed concurrently (i.e. SOCCA objects can be multi-threaded). Thus, intra-object method use necessitates coordination between threads. To emphasise the consequences of intra-object method use, it is made explicit in the data perspective through the uses relationship.

Because of the amount of information involved, the class diagram is often split into several sub diagrams, such as import/export diagrams, class diagrams without relationships, inheritance diagrams and aggregation diagrams. Such a split often has an informal meaning to the modeller, but is for the purposes of this formalisation merely a matter of convenient representation; it has itself no formal semantics. The uses relationship, which focuses on the method import necessary for communication, is often drawn in such a separate import/export diagram.

Example In Figure 1, we show a fragment of a import/export diagram where class c imports method $M1$ from class d. This import indicates class c can call this method of d. Method $M2$ needs to be imported separately if class c is to be able to call it. □

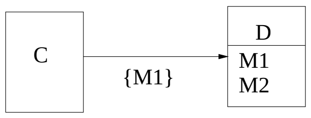


Figure 1: A class diagram with a uses relationship

With the uses relationship covered, we have discussed the parts of the data perspective relevant for modelling communication. The next section continues with the behaviour perspective.

2.2 Behaviour Perspective

The *behaviour perspective* deals with visible behaviour (behaviour that is visible to other classes); whereas the *functionality perspective* describes hidden behaviour (which describes the functionality of the various methods). Later on, in the communication perspective, we will describe the coordination between the behaviours of objects.

The behavioural aspects of SOCCA models are specified through State Transition Diagrams (STDs): graphical diagrams containing states and labelled transitions between them.

With respect to the behaviour perspective, we associate with each class one or more STDs. These STDs are called *external* STDs and specify the *external behaviour* of the class. This is the order in which calls to methods the class exports are accepted. The traversals through the external STD specify in what order the methods of the class can be successfully called. The ϵ transitions in the STD can always be made in such a traversal.

Example In Figure 2, we show the external STD of class d of Figure 1. Each state is drawn as a circle and the initial state is marked with an arrow with a non-shaded point. Each edge of the STD is either an ϵ transitions or is labelled with a method name. The external STD of d specifies that the iterative calls to methods $M1$ and $M2$ will be accepted. So, the sequence of calls $\langle M1, M2, M1, M2, M1\rangle$ will be accepted while the sequence $\langle M1, M2, M2\rangle$ will not occur. $\square$

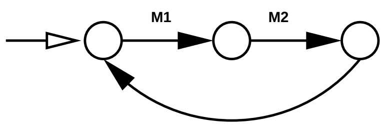


Figure 2: The external STD of a class

The next section continues with the functionality perspective.

2.3 Functionality Perspective

With each method of a class, we associate an *internal behaviour STD* that describes how that method is realised. Each transition of internal behaviour STD can be labelled with a method call. Calls can however only be made to methods which are imported as defined by the uses relationships. A method m of a class c can only make a call to a method m' of class d if method m' is a label of a uses relationship from c to d.

The transitions in an internal behaviour STD may be labelled with "*act method name*" (indicating activation of the execution of *method name*) or "*call class.method name*" (indicating a request to start the execution of the method).

Example In figure 3 we give a typical internal STD (belonging to a method *Foo*), which makes two calls (one, *B.bar* of a class *B*; the other to *C.Baz* of a class *C*) and does some internal stuff (an unlabelled (ϵ) transition).

Rather than using separate end states (a state filled with a smaller circle), an ϵ transition from what is effectively an end state to the initial state, which is also an end state, is provided. This convention is used in several SOCCA publications. The underlying intuition is that of a process that in some sense becomes dormant after handling a call and is woken up by a new call. This behaviour is defined in the communication perspective.

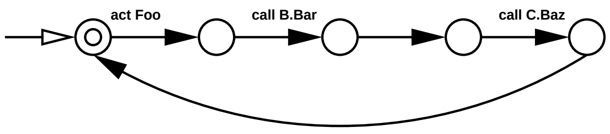


Figure 3: A typical internal STD

□

The next section continues with the explanation of the communication perspective.

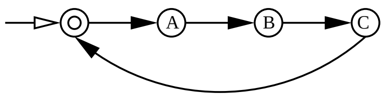
2.4 Communication Perspective

The communication perspective is where SOCCA differs the most from other object oriented modelling languages and where the difficulties for readers new to SOCCA usually really begin. This perspective regulates the coordination of the behaviour of methods.

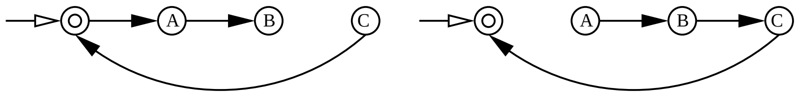
With respect to the functionality perspective, each internal STD of a method defines the full potential behaviour of a method. With respect to the communication perspective, subprocesses are used to *temporarily* restrict the behaviour of these internal STDs. The desired coordination between classes is achieved by ensuring that the right subprocesses are prescribed to the right internal STDs.

A subprocess of an STD is another STD where this new STD has a subset of the states and transitions of the original. The current possible behaviour of an internal STD is defined as the total possible behaviour defined by the STD restricted by all the subprocesses which are currently prescribed.

Example In Figures 4(a) and 4(b) we give two subprocesses of the internal STD of Figure 2.4. The labels of the STD are left out to keep things simple. By alternatively prescribing either subprocess $R1$ or $R2$ we can (temporarily) restrict the possible behaviour of the internal STD. For example, if we first prescribe subprocess $R1$, then the state labelled with "C" is not reachable from the state labelled with "B" of the internal STD. We prescribe subprocesses $R2$ when this situation is no longer desirable. □



Prescribing the right subprocesses in SOCCA over time for the internal STDs achieves the desired synchronisation in the communication between the classes and objects. For example, an edge is taken as a transition of an internal STD and the edge is labelled with a call to a method of a class. The STD is not allowed to proceed beyond the reached state with



(a) Subprocess R1 (b) Subprocess R2

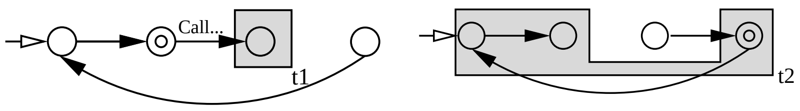
Figure 4: An STD with two subprocesses

further calls by prescribing the appropriate subprocess until the desired synchronisation constraints are met.

Example In Figure 2.4, consider the case where the edge from the state A of the STD to state B is labelled with a call. We can let the STD "wait" after the call by prescribing subprocess $R1$ of Figure 4(a). We can let the STD proceed by prescribing subprocess $R2$ instead. □

At this point we are ready to introduce the concept of a *trap*. A *trap* defines the part of a subprocess where coordination is desired. A trap is a subset of the states of a subprocess where no state not in the trap can be reached. The trap can not be left by taking one or more transitions; thus the name "trap".

Example In Figures 5(a) and 5(b) we once more show the subprocesses 4(a) and 4(b) but now with an additional label of a call for 5(a) and a trap for both subprocesses. The traps are shown as shaded areas. When more than one trap is presented with a subprocess, they are often given names, in this case $t1$ and $t2$. These two subprocesses are used to provide the means to synchronise the call by alternatively prescribing the subprocesses. Trap $t1$ indicates the call has been made and the internal STD is "waiting" to proceed. The internal STD can proceed if subprocess $R2$ is prescribed instead. Trap $t2$ indicates the switch has been made. We can then switch back to subprocess $R1$ and wait for another call which has to be synchronised. □



(a) Subprocess R1 (b) Subprocess R2

Figure 5: An STD with two subprocesses

Up to now we have left the reader in the dark as how the subprocesses for the internal STDs are prescribed. This is done by *managers* which determine which subprocesses are prescribed for an internal STD at each point in time. Each class has one (or more) managers which are defined as an external STD of the class with extra states and labels. Managers are yet another special STD. Each state of the manager STD is labelled with the subprocesses which are prescribed when the manager is in that state. A switch in state of the manager thus results in a possible change in the subprocesses prescribed by the manager. The edges of the labels of the manager are labelled with the names of traps. These traps must have been reached in the appropriate subprocesses if the transition is to be taken.

Example Figure 6 gives an example manager. This manager coordinates the internal STD of Figure 2.4 by prescribing subprocesses $R1$ and $R2$ when traps $t1$ and $t2$ are reached. The subprocesses $R1$ and $R2$ are used to coordinate a call and traps $t1$ and $t2$ are used to detect when a subprocess switch is necessary. The activities of the STD of Figure 2.4 are also coordinated with another internal STD with subprocesses $E1$ and $E2$ with traps $t3$ and $t4$, all of which are not shown here. The example however shows that the progress of our internal STD is dependent on the progress of the "omitted" internal STD and whether it reaches traps $t3$ and $t4$. □

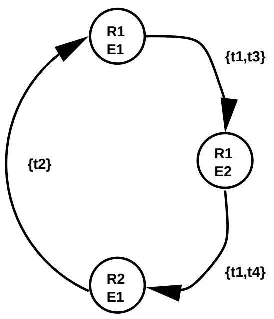


Figure 6: A manager

We call each internal STD for which subprocesses are prescribed by a manager an *employee* of that manager. A manager typically has two types of employees. First of all, the manager of a class has as employees the internal STDs of its own class. The manager regulates when a method of the class can activate by taking the transition labelled with "act". The method of the class then goes from a dormant state to an *act*ive state. Secondly, a manager has as employees the internal STDs which make a call to a method of the class to which the manager belongs. The manager ensures by prescribing the right subprocesses that the calling internal STD cannot proceed until the desired synchronisation is achieved.

The notion of subprocess, trap, manager and employee are the innovative parts of SOCCA and were originally introduced in PARADIGM [Gro88].

3 Concluding Remarks

We have seen how the basic communication between classes/objects is regulated in SOCCA and we have covered the necessary SOCCA concepts. The *uses* relationship defines what methods are imported. The *external* STDs of a class regulate the order in which the methods of a class can be called. *Internal* STDs define the behaviour of the methods. *Managers* prescribe *subprocesses* to their *employees* and wait for *traps* to prescribe new subprocesses to their employees.

We have however simplified many issues in this text. We have, for example, not discussed inheritance or aggregation. We have furthermore not discussed the subtle effects which arise when one internal STD is prescribed subprocesses by more than one manager. Nor have we discussed the SOCCA concepts at a very formal level or demonstrated the modelling techniques with a large example. This was not our goal. Our intention was to write a simple introduction to the SOCCA concepts to give a reader new to SOCCA a means to reduce the steep learning curve experienced with SOCCA publications. The reader of this text should now be better prepared to tackle a "real" SOCCA publication. See "http://www.wi.LeidenUniv.nl:/CS/SEIS/socca-bib.html" for a complete up to date electronic list with downloads.

References

[DGSK$^+$99] J.H.M. Dassen, L.P.J. Groenewegen, I.G. Sprinkhuizen-Kuyper, P.W.M. Koopman, P.J. 't Hoen, and G. Engels. A formalisation of SOCCA using Z; part 1: the type level concepts. Technical Report 1999–03, Leiden Institute of Advanced Computer Science, February 1999. Available on the web as `http://www.wi.leidenuniv.nl/TechRep/1999/tr99-03.ps.gz`.

[EG94] Gregor Engels and Luuk P.J. Groenewegen. SOCCA: Specifications of coordinated and cooperative activities. In A. Finkelstein, J. Kramer, and B.A. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 71–102. Research Studies Press Ltd. / John Wiley & Sons Inc., 1994. Taunton 1994.

[Gro88] L.P.J. Groenewegen. Parallel phenomena, 1986–88. A series of technical reports, consisting of [Gro86, Gro87c, Gro87g, Gro87h, Gro87i, Gro87b, Gro87e, Gro87d, Gro87f, Gro88a, Gro88b, Gro87a].

[Gro86] L.P.J. Groenewegen. Processes. Technical Report 86-20, Department of Computer Science, Leiden University, 1986. Part of [Gro88].

[Gro87a] L.P.J. Groenewegen. Changing managing cooperation in a hierarchy. Technical Report 88-18, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87b] L.P.J. Groenewegen. A critical section model. Technical Report 87-18, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87c] L.P.J. Groenewegen. Decision processes. Technical Report 87-01, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87d] L.P.J. Groenewegen. Dijkstra's semaphore solution. Technical Report 87-29, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87e] L.P.J. Groenewegen. Goeman's solution and a stochastic solution. Technical Report 87-21, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87f] L.P.J. Groenewegen. Lamport's bakery problem. Technical Report 87-32, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87g] L.P.J. Groenewegen. Modelling. Technical Report 87-05, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87h] L.P.J. Groenewegen. Parallel processes. Technical Report 87-06, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87i] L.P.J. Groenewegen. Two examples of a parallel control process. Technical Report 87-11, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro88a] L.P.J. Groenewegen. Trap process hierarchy: an almighty manager. Technical Report 88-15, Department of Computer Science, Leiden University, 1988. Part of [Gro88].

[Gro88b] L.P.J. Groenewegen. Trap process hierarchy: cooperating managers. Technical Report 88-17, Department of Computer Science, Leiden University, 1988. Part of [Gro88].

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., 1991.

[tHDG+99] P.J. 't Hoen, J.H.M. Dassen, L.P.J. Groenewegen, I.G. Sprinkhuizen-Kuyper, P.W.M. Koopman, and G. Engels. SOCCA extended with UML like packages. Technical Report 99-06, liacs, September 1999.

[UML99] Unified modeling language 1.3. Technical report, Rational Software Corporation, 1999.