Formalising Object-Oriented Modelling Languages

J.H.M. Dassen[1*], L.P.J. Groenewegen[1], G. Engels[2], P.J. 't Hoen[1], P.W.M. Koopman[3], and I.G. Sprinkhuizen-Kuyper[4]

[1] Leiden University, Leiden Institute of Advanced Computer Science, P.O. Box 9512, 2300 RA Leiden, The Netherlands
[2] University of Paderborn, Dept. of Computer Science, D-30095 Paderborn, Germany
[3] University of Nijmegen, Dept. of Computer Science, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.
[4] Universiteit Maastricht, Institute for Knowledge and Agents Technology, P.O. Box 616, 6200 MD Maastricht, The Netherlands

Abstract. The semantics of object-oriented modelling languages is often only given by informal means, such as text in a natural language. This makes object-oriented models developed in these modelling languages less precise, clear and unambiguous than is desirable.
We show how the static semantics of an advanced object-oriented language can be defined accurately. This formal definition of the language strongly increases the value of the models developed in this language: the meaning of these models is now unambiguously defined.
We use a separate language for the formal specification of the language in order to avoid the known problems with defining the semantics of a language in the language itself (such a recursive definition introduces an unknown amount of poorly defined concepts).
The structure of the formalisation employs a meta level, rather than a transformational approach. This eliminates issues regarding the semantics of a target language and yields a more abstract and rigorous specification which is also better suited for use in the development of tools that employ the modelling language. We use the specification language Z as it has a high abstraction level, is widely used, and has a sound mathematical foundation. Additionally, there is a standard definition of Z, and tool support is available. Tools support has proven itself to be valuable.
We use SOCCA as the object-oriented modelling language since it has very powerful capabilities to express and control coordination and communication between objects. We give an intuitive description of SOCCA's approach to communication and discuss the formalisation of its static semantics.

1 Why formalise object-oriented modelling languages?

The primary goal in developing a modelling language is to make it possible to model particular situations in precise, clear and unambiguous terms. For this, it is important that the modelling language itself be precise, clear and unambiguous, powerful and yet still easy to use. The modelling language is a vehicle for communication between and among the modellers, clients and implementors; the concepts it supports influence how modellers do their job.

An essential requirement for the modelling language to be clear is that the modellers (and automated agents involved) have identical notions of the semantics of the modelling language (and thus of individual models). The availability of a definite description of both syntax and semantics of the language is essential.

Nowadays there is increased interest in rigorously defining the semantics of modelling languages using formal specification languages, for several reasons. One is that tool support for

* Please use the primary author's contact address: jdassen@wi.LeidenUniv.nl

a modelling language is a necessity for industry acceptance of a modelling language. Industry relies heavily on the availability of tools throughout a large part of the software life cycle. Interoperability between tools is necessary in such a context. A formal specification of the modelling language can play an important role in achieving this interoperability.

Another reason is the current push to standardise on UML as a modelling language. UML is quite large (as it is essentially a union of several predecessor modelling languages), and not very well understood yet. A clear semantics is necessary to prevent splintering and tool interoperability problems due to different interpretations.

Finally, modelling languages are interesting test cases for the formal methods community. Formal methods have matured over recent years, and in the formal methods community there is a feeling that formal methods have now reached sufficient maturity to be capable of spreading beyond the fairly small number of application domains that currently employ them [NAS95, C$^+$97].

In this paper, we use the object-oriented modelling language SOCCA, which we prefer for its expressive power for modelling communication and coordination between object. In addition to the general reasons for formalising object-oriented modelling languages, there are SOCCA-specific ones. SOCCA's approach to modelling communication and concurrency, discussed in Section 6, is based on PARADIGM [GSO86], a formalism rooted in the mathematical framework of semi-Markov decision processes. So far, PARADIGM has mostly been described in these mathematical terms, making acceptance in Computer Science difficult. The formalisation effort provides an opportunity to do away with PARADIGM specifics that have become legacy in the context of SOCCA, and allows us to re-phrase PARADIGM as used in SOCCA's communication perspective using concepts more common in Computer Science, such a State Transition Diagrams (STDs).

First we discuss the what and how of formalising an object-oriented modelling language and our choice of formal specification language. Then we look at the formalisation of our object-oriented modelling language, SOCCA reported on in our report [DGSK$^+$99].

2 What is to be formalised?

It is our goal to describe both the underlying context-sensitive syntax (static semantics) and the (dynamic) semantics of SOCCA using a single formalism. Of this goal, the first part has now been realised: in our report [DGSK$^+$99] we give the static semantics of SOCCA in the form of Z schemata. This provides the basis for realising the second part of our goal: the description of the dynamic semantics of SOCCA.

In this paper, we motivate design decisions made in the formalisation, provide an overview of the formalisation, and highlight some key parts.

3 How to formalise object-oriented modelling languages

Several approaches to formalising the semantics of object-oriented modelling languages appear in the literature. It is interesting to look at the choices we made for our SOCCA formalisation along the following lines:

- The language being formalised.
- The language in which the semantics is expressed.
- The level at which the formalisation is carried out.

We do this by putting our approach in the context of the literature.

UML's metaclass diagram and OCL. UML's authors have expressed UML's semantics in UML itself augmented with a logic notation. More rigour and formality is required of a good semantics. The limitation of this "reflexive" approach to formalisation is that it does not force one to step outside the concepts of the language being formalised. Formalisation into a language based on a different paradigm does force one to pay more attention to the underlying concepts of the modelling language.

Semantics in concurrency formalisms. SOCCA has a strong emphasis on precise modelling of concurrent behaviour and communication. One might argue that formalising into a formalism more geared towards concurrency than Z is appropriate. The use of such formalisms, like Petri Nets, CCS and CSP might simplify the effort in formalising these important aspects of SOCCA. There are two reasons why we do not consider them ideally suited for formalising SOCCA.

The first is that SOCCA's communication perspective, being based on PARADIGM, offers concepts not readily available in other concurrency formalisms (we describe these concepts in Section 6). Thus, a direct formalisation of SOCCA's communication perspective into a concurrency formalism is cumbersome. Z, being a more general purpose specification formalism, allows us to express SOCCA's communication concepts in a clearer way, as it does not force us to fit them in another concurrency approach's mould. As the communication concepts are where SOCCA differs most from other approaches, this is quite important.

The second reason is that concurrency formalisms are less generic than Z. While they are perhaps suitable for formalisation of the communication perspective of SOCCA, employing them would make it quite difficult to address the other perspectives of SOCCA, which is necessary to provide an integrated semantics of the whole of SOCCA.

Thus, characteristics of the modelling language being formalised should be taken into account for the choice of specification formalism.

The level of the formalisation. [EA98] identifies two levels of abstraction for a formalisation of a modelling language: through a *translational approach* or at the *meta level*.

In the translational approach, the focus is on how to translate individual models to specifications (e.g. by translating each class to a Z schema).

In the meta level approach, the focus is on translating a language to a single specification, rather than models each to a specification of their own. This approach can be extended to develop a translational approach.

In formalising SOCCA, we have chosen the meta level approach: we focus on formalising the SOCCA language itself, rather than on formalising SOCCA models. This yields a more abstract semantics which may be more difficult to use with generic tools, but which is more rigorous and better suited as a basis for the development of SOCCA specific tools.

4 The formal specification language Z

The formalisation of SOCCA is being carried out in the formal specification language Z [Spi92, ZZads].

Several factors positively influenced our choice of Z as the specification language, including its abstraction level, widespread use, strong mathematical foundation, level of standardisation and availability of tool support.

5 The structure of the SOCCA modelling language

Although object-oriented modelling has proven itself to be a useful practical approach to tackling the issues of analysing, designing, specifying and implementing complex software systems, there is still room for improvement. One of the ways in which we want to improve object-oriented modelling is in the description of and control over the communication of objects executing in parallel.

SOCCA (Specification of Coordinated and Cooperative Activities) [EG94] is both a language and a method for object-oriented modelling. Here we deal only with the SOCCA language. It is *eclectic*: it combines proven features from other object-oriented modelling languages in several *perspectives* that address different aspects of a model. In addition to a *data perspective, behaviour perspective* and a *functionality perspective* that use concepts well-known from languages like OMT and UML, it has a *communication perspective* that provides a powerful mechanism to describe and control communication between objects (see e.g. [EGK96, EGK99]). This ability to model coordinated behaviour to any desired degree is of particular importance for use in the domain of workflow modelling. The communication concepts in this communication perspective are extensions of the concepts of state transition diagrams and state transition machines that are used in the behaviour and functionality perspectives.

SOCCA has been used in several industry projects and is the subject of ongoing research; one of the research projects works towards capturing the syntax and the semantics of SOCCA (and thereby, of SOCCA models) in a formal specification method.

Currently, we have completed the formalisation of the type level of SOCCA [DGSK+99], i.e. we have given a context-sensitive syntax of the SOCCA language through Z schemata that captures its static semantics (i.e. what are syntactically valid SOCCA models).

We have formalised the concepts of each of SOCCA's perspectives:

The data perspective which focuses on the static, structural aspects of models with concepts like attributes, methods, classes, relationships, inheritance, aggregation, uses relationship and binding.

The behaviour perspective which focuses on dynamic aspects of individual classes and objects which are made available to other classes and objects through the concept of external behaviour STDs.

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

The communication perspective which focuses on the communication between individual classes and objects through the notions of subprocess, trap, partition and trap structure,

employee process and manager process. In Section 6 we describe these notions in an intuitive way, and in Section 7 we show part of their formalisation.

Also, we have described how the perspectives relate to each other. For instance, we have described how managers in the communication perspective are based on external STDs from the behaviour perspective.

In the next sections, we will give an intuitive description of SOCCA's most interesting feature, its communication perspective, and show part of its formalisation.

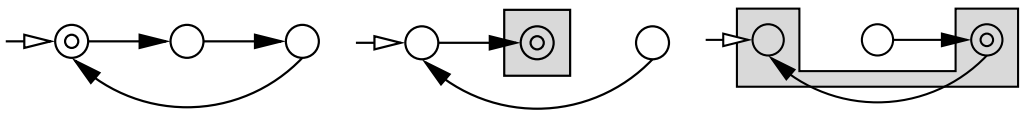
6 SOCCA's communication perspective: intuitive description

The communication perspective is where SOCCA differs the most from other object-oriented modelling languages. Here we sketch the intuition underlying the communication perspective and its formalisation.

A fundamental observation about communicating processes is that their behaviour can be viewed as having two levels. The first is the level of *local behaviour* which describes the pieces of behaviour that the process may have which do not require communication with other processes. Such local behaviour has parts in which no communication is desired, and no coordination is necessary, and parts in which communication is desired to arrange coordination to prepare the way for another piece of local behaviour. Until this communication has taken place, the process is restricted to the current piece of local behaviour. The second, more abstract, level is that of *global behaviour* which describes how the processes' behaviour may be switched from one piece of local behaviour to another through coordination by communication.

In SOCCA we describe the global behaviour of classes through an external STD, and the local behaviour of methods through internal STDs. The different parts of local behaviour we describe by *subprocesses* and *traps*. A subprocess describes a temporary restriction of behaviour, a piece of local behaviour. A *trap* defines the final part of a subprocess where coordination is desired; it is a set of states that cannot be left within the subprocess.

Example 1. In figure 1, a simplified STD is shown (labels are left out), together with two possible subprocesses and some of their traps (shown as shaded areas). When more than one trap is present within a subprocess, they are given numbers. The subprocesses are partial versions of the original STD (disregarding initial and final states).



(a) Full STD (b) Subprocess 1 (c) Subprocess 2

Fig. 1. An STD with two subprocesses

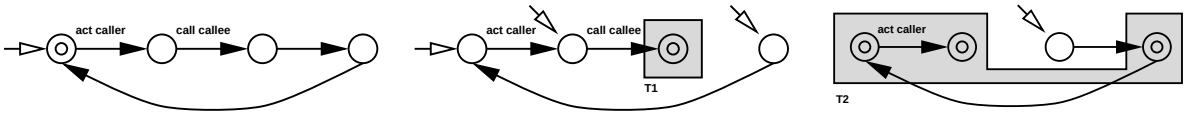
In light of communication, we distinguish two roles of STDs: *employee* and *manager*. An employee is an STD augmented by a structure of subprocesses and traps known as a *partition and trap structure*. It is managed by a manager (meaning the manager prescribes which transitions between its subprocesses an employee may do, and when it may do so). The manager is an STD augmented with two functions: the *state interpreter*, which maps its states to prescribed subprocesses, and the *transition interpreter*, which labels its transitions with traps that its employees must have reached for the transition to be allowed.

In SOCCA, the external STDs form the basis for the managers, and the internal STDs for the employees.

There is a behavioural consistency that works in both directions: an employee's behaviour obeys the restriction imposed by the current subprocess prescribed by the manager, while the manager's behaviour obeys the restrictions imposed by the subprocesses of its employees (not making a transition labelled with a trap that has not been reached yet).

By itself, PARADIGM lacks the structure provided by the object orientation of SOCCA and thus allows the modeller very large degrees of freedom. In SOCCA this freedom has been restricted through the object oriented structure, making it more manageable. In SOCCA, the modeller no longer has the freedom of choosing employee and manager roles arbitrarily: a class' external STD(s) gets the role of manager of the internal STD(s): the external STDs receive messages (calls) and start up behaviours of internal STDs to handle them.

Example 2. As an illustration of how the communication perspective in SOCCA is used, consider the following situation: we have two classes, A and B. Method $A.Caller$ needs to perform a synchronised call to method $B.Callee$, i.e. it calls *Callee* and has to wait until that call has been handled completely.



(a) STD (b) First subprocess R_1 (c) Second subprocess R_2

Fig. 2. Caller

Caller's STD is depicted in figure 2(a). It has a fairly simple structure: activation, call *Callee*, some internal stuff, and repeat when desired.

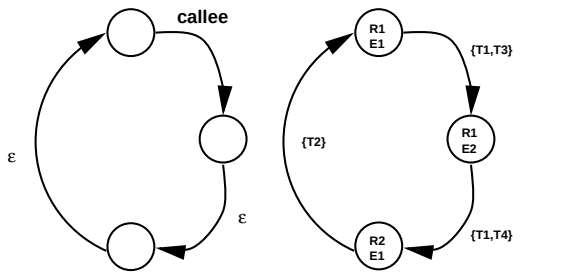
The handling of the call to *Callee* induces two subprocesses: one, **R1** (depicted in figure 2(b)) in which the actual call is allowed and in which the trap **T1** expresses waiting for the call to finish; the other, **R2** (depicted in figure 2(c)) in which permission to perform the call is temporarily revoked; its big trap **T2** indicating its willingness to regain that permission as soon as possible.

Callee's structure is more simple than *Caller*'s: activation, and internal stuff (see figure 3(a)). Like in *Caller*, the synchronised way we want to call it induces two subprocesses: the first, **E1**

(a) STD (b) First subprocess E_1 (c) Second subprocess E_2

Fig. 3. Callee

with trap **T3** (in figure 3(b)) in which *Callee* waits to perform its activities; the second, **E2** with trap **T4** (in figure 3(c)) in which *Callee* performs them.



(a) External STD of A (b) Corresponding manager

Fig. 4. External STD and corresponding manager

In figure 4, a suitable manager is depicted. The state and transition interpreters are indicated by an appropriate labelling of the states and transitions respectively.

7 SOCCA's communication perspective: formal description

In this section, we take a look at a part of the formalisation of the communication perspective; the full details are in [DGSK+99].

We start by defining the infix relationship *isSubProcessOf*: an STD *subp* is a subprocess of an STD *std* if it can be obtained from *std* by removing some states and labels. (A schema for STD, having member variables *states* some of which are *initial* or *final*, *labels*, and a transition relation *transrel* is assumed.)

$$isSubProcessOf : STD \leftrightarrow STD$$

$$\forall std, subp : STD \bullet subp \ \underline{isSubProcessOf} \ std \Leftrightarrow$$
$$subp.states \subseteq std.states \land$$
$$subp.labels \subseteq std.labels \land$$
$$subp.transrel \subseteq std.transrel \cap ((subp.states \times subp.labels) \times subp.states)$$

Next we define $isTrapOf$: a set of states of an STD is a trap if, once reached, it cannot be left within that STD.

$$
\begin{array}{|l}
isTrapOf : \mathbb{P}\, STATE \leftrightarrow STD \\
\hline
\forall S : \mathbb{P}\, STATE \bullet \forall std : STD \bullet \\
\quad S \ \underline{isTrapOf}\ std \Leftrightarrow \\
\quad\quad S \subseteq std.states \land S \neq \varnothing \land \\
\quad\quad (\forall s, t : std.states \bullet \\
\quad\quad\quad \forall l : std.labels \mid s \in S \land ((s, l), t) \in std.transrel \bullet t \in S)
\end{array}
$$

Using these, we define a relation $isTrapConnectionOf$ which describes whether a particular set of states is a trap that connects two subprocesses.

$$
\begin{array}{|l}
isTrapConnectionOf : STD \times \mathbb{P}\, STATE \times STD \leftrightarrow STD \\
\hline
\forall std, subp_1, subp_2 : STD;\ trap : \mathbb{P}\, STATE \bullet \\
\quad (subp_1, trap, subp_2)\ \underline{isTrapConnectionOf}\ std \Leftrightarrow \\
\quad\quad (subp_1\ \underline{isSubProcessOf}\ std \land \\
\quad\quad subp_2\ \underline{isSubProcessOf}\ std \land \\
\quad\quad trap\ \underline{isTrapOf}\ subp_1 \land \\
\quad\quad ((subp_1 = subp_2) \lor (trap \subseteq subp_2.initial \land trap \subseteq subp_1.final)))
\end{array}
$$

Now we can define what a partition and trap structure is: an STD with a set of subprocesses, each with a set of traps such that the subprocesses together cover the STD, and are locally connected through traps.

$$
\begin{array}{|l}
isPartitionAndTrapStructureOf : (\mathbb{P}(STD \times \mathbb{P}(\mathbb{P}\, STATE))) \leftrightarrow STD \\
\hline
\forall part : \mathbb{P}(STD \times \mathbb{P}(\mathbb{P}\, STATE));\ std : STD \bullet \\
\quad (part, std) \in isPartitionAndTrapStructureOf \Leftrightarrow \\
\quad\quad (std.states = \{state : STATE \mid \exists\, partstd : part \bullet state \in (first\ partstd).states\}) \land \\
\quad\quad (std.transrel = \\
\quad\quad\quad \bigcup \{trans : (STATE \times SYMBOL) \leftrightarrow STATE \mid \exists\, partstd : (STD \times \mathbb{P}(\mathbb{P}\, STATE)) \bullet \\
\quad\quad\quad\quad trans = (first\ partstd).transrel\}) \land \\
\quad\quad (\forall subp : STD;\ traps : \mathbb{P}(\mathbb{P}\, STATE) \mid (subp, traps) \in part \bullet \\
\quad\quad\quad subp\ \ \underline{isSubProcessOf}\ std \land \\
\quad\quad\quad (\forall trap : \mathbb{P}\, STATE \mid trap \in traps \bullet \\
\quad\quad\quad\quad trap\ \underline{isTrapOf}\ subp \land \\
\quad\quad\quad\quad (\exists\, subp_2 : STD;\ traps_2 : \mathbb{P}(\mathbb{P}\, STATE) \mid \\
\quad\quad\quad\quad\quad (subp_2, traps_2) \in part \bullet \\
\quad\quad\quad\quad\quad (\exists\, ctrap : traps_2 \bullet \\
\quad\quad\quad\quad\quad\quad (subp, ctrap, subp_2)\ \underline{isTrapConnectionOf}\ std))))
\end{array}
$$

An STD, together with a suitable partition and trap structure can play the role of employee.

employee

$$std : STD$$
$$pts : \mathbb{P}(STD \times (\mathbb{P}(\mathbb{P}\, STATE)))$$

$$pts \;\; \underline{isPartitionAndTrapStructureOf} \;\; std$$

To make the definition of manager manageable, we introduce abbreviations for the types of state and transition interpreters.

$$stateint == STATE \pfun STD$$
$$transint == (STATE \times SYMBOL) \times STATE \pfun (\mathbb{P}\, STATE)$$

Using these abbreviations, we define manager. A manager is an STD combined with a collection of employees, and state and transition interpreters for each of them.

manager

$$std : STD$$
$$empsti : \mathrm{seq}(employee \times stateint \times transint)$$

$$\begin{aligned}
&(\forall\, i : 1 \,..\, \#empsti \bullet \\
&\qquad (\forall\, e : employee;\; si : stateint;\; ti : transint \mid (e, si, ti) = empsti\ i \bullet \\
&\mathrm{dom}\ si = std.states \land \\
&\mathrm{dom}\ ti = \{t : (STATE \times SYMBOL) \times STATE \mid \\
&\qquad\qquad \exists\, s_1, s_2 : std.states;\; sym : SYMBOL \bullet \\
&\qquad\qquad\qquad t = ((s_1, sym), s_2) \land t \in std.transrel\} \land \\
&\qquad (\forall\, s_1, s_2 : std.states;\; sym : std.labels \mid \\
&\qquad\qquad ((s_1, sym), s_2) \in std.transrel \bullet \\
&\qquad\qquad si\ s_1\ \underline{isSubProcessOf}\ e.std \land \\
&\qquad\qquad si\ s_2\ \underline{isSubProcessOf}\ e.std \land \\
&\qquad\qquad ti\ ((s_1, sym), s_2)\ \underline{isTrapOf}\ si\ s_1 \land \\
&\qquad\qquad (si\ s_1, ti\ ((s_1, sym), s_2), si\ s_2)\ \underline{isTrapConnectionOf}\ e.std)))
\end{aligned}$$

It is interesting to study the relationship between the formalisation given in this section, and the informal description in the previous section. It is clear that the formalisation cannot be understood without the natural language explanation, but that the formal text is much more precise.

8 Results of the formalisation

The main result of the first phase of the formalisation effort is the static semantics of SOCCA in Z ([DGSK$^+$99]), a definite description of what are correct SOCCA models.

The process of formalisation so far additionally has brought us new insights into the structure of the SOCCA language; here we discuss several of them.

Binding. The formalisation has highlighted the concept of binding which captures the meaning of polymorphism by inheritance, the similarities between methods and attributes (unified through the concept of feature).

Structure of the SOCCA language. The formalisation also shows a way to structure an explanation of SOCCA's concepts. Z's "no forward references" nature forced us to write the formal text without forward references; the order this imposes allows us to structure the informal text (the natural language description of SOCCA's concepts) so as to contain but a few forward references. As the previous section illustrates, complex concepts are constructed from simpler ones in bottom-up fashion.

Maturation of the SOCCA language. The formalisation effort also forced us to make numerous minor and some fundamental decisions about the SOCCA core language such as whether to have multiple external STDs per class (yes, as it makes parallelism within objects more explicit) and what visibility mechanism to use (a crude one, but one which practical experience has shown to be quite powerful).

In our experience developing the semantics of an object-oriented language forces one to focus on both the details and the whole picture of the language, and improves one's understanding of the choices involved in a modelling language and their consequences. To address real world modelling needs, practical object-oriented specification languages mix concepts and elements from different areas rather than imposing a single vision on the modeller. The mix needs to form an integrated whole for it to be usable in practice. Formalising such a language forces its designers to study carefully the interactions between the elements in the mix and as such should play an important part in the development and maturation of an object-oriented specification language.

Tool support for Z. We found that even our limited use of tool support (using Z/EVES only as an automated type checker) was helpful in developing the abstract syntax of SOCCA and in capturing its internal consistency requirements.

9 Current research and future work

The instance level. Current research in the formalisation effort is focusing on the instance level semantics of SOCCA. The instance level is the level at which individual "runs" of SOCCA model instances can be described, and at which the full semantics of SOCCA's type level can be expressed:

- What do transitions mean?
- How are instances (of classes, relationships, state transition diagrams, managers and employees) created and destroyed?
- What concurrent actions are possible?

– Which object can call what method (runtime encapsulation)?

One of the topics is whether or not a standardised notation for the instance level is necessary, or whether it suffices to have an instance level in the formalisation, without a concrete notation to illustrate it.

Prototypical instance layer. We expect to extend SOCCA with a *prototypical instance layer* (cf. the notion of a prototypical object level [JBAG97]) based on the instance level which can express requirements about groups of objects; such requirements cannot be expressed at the existing type level.

Other SOCCA research. Outside the formalisation effort, research is done on developing a module concept which will make the SOCCA approach scale to larger problems and on the possibility of expressing patterns ([CS95]) in SOCCA and developing patterns for SOCCA; see [Bru98]. We expect this work to benefit from the formalisation of the existing SOCCA concepts, and expect its semantics to be expressed as an extension/reworking of the core SOCCA semantics.

Other SOCCA research deals with software process modelling [DKW98] and evolution.

10 Discussion

In this paper we have argued the need for formalisation of the syntax and the static and dynamic semantics of object-oriented modelling languages. We have looked at some of the important choices that need to be made in the process of developing such a formalisation, such as the choice of specification formalism and the level of the formalisation (translational approach or meta level) and we explained our decisions with regard to them (Z as specification formalism; formalisation at the meta level). We have given an overview of SOCCA, an object-oriented modelling language which has strong expressive power in the field of communication and control of concurrency. We have given an intuitive description of SOCCA's communication perspective, which provides this expressive power. We have shown how concepts from the communication perspective are expressed in the formal specification language Z. We have discussed some of the insights obtained by the development of the specification of SOCCA's static semantics in Z ([DGSK$^+$99]), such as the role of binding, the possibility of a forward-references-free description of SOCCA. We stress the importance of using formalisation techniques in language development as a means of achieving language maturation. As future work we will formalise the instance level of SOCCA to express SOCCA's the dynamic semantics.

References

[Bru98] H.G. Brugman. Software process modeling in SOCCA. Master's thesis, Department of Computer Science, Leiden University, May 1998.

[C$^+$97] Judith Crow et al. *NASA Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume II: A Practitioner's Companion.* NASA Office of Safety and Mission Assurance, Washington, DC, 1997. Available at `http://eis.jpl.nasa.gov/quality/Formal_Methods/`.

[CS95] James O. Coplien and Douglas C. Schmidt. *Pattern Languages of Program Design.* Addison-Wesley, Reading, Mass., 1995.

[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`.

[DKW98] Jean-Claude Derniame, Badara Ali Kaba, and David Wastell, editors. *Software Process: Principles, Methodology and Technology*, number 1500 in Springer Lecture Notes in Computer Science. Springer Verlag, Berlin, Heidelberg, New York., 1998.

[EA98] A. S. Evans and A.N.Clark. Foundations of the unified modeling language. In *2nd Northern Formal Methods Workshop, Ilkley*, electronic Workshops in Computing. Springer-Verlag, 1998.

[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.

[EGK96] G. Engels, L.P.J. Groenewegen, and G. Kappel. Object-oriented specification of coordinated collaboration. In Noboyoshi Terashima and Edward Altman, editors, *Proceedings IFIP World Conference on IT Tools, 2-6 September 1996 — Advanced IT Tools.*, pages 437–449, Canberra, Australia, September 1996. Chapman & Hall, London, Wienheim, New York, Tokyo, Melbourne, Madras. Also available as Technical Report 96-24 Department of Computer Science, Leiden University.

[EGK99] G. Engels, L.P.J. Groenewegen, and G. Kappel. Coordinated collaboration of objects. In M. Papazoglou, St. Spaccapietra, and Z. Tari, editors, *Object-Oriented Data Modelling Themes*. MIT press, 1999. To appear.

[GSO86] L.P.J. Groenewegen, M.R. van Steen, and G. Oosting. Modelling parallel phenomena. In Vansteenkisten et al., editor, *Proceedings 2nd European Simulation Congress*, pages 45–51, Antwerp, Belgium, September 1986.

[JBAG97] Stefan Joos, Stefan Berner, Martin Arnold, and Martin Glinz. Hierarchische Zerlegung in objektorientierten Spezifikationsmethoden. *Softwaretechnik-Trends*, 17(1):29–37, February 1997.

[NAS95] NASA Office of Safety and Mission Assurance, Washington, DC. *NASA Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion*, 1995.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[ZZads] Z archive. URL: `http://www.comlab.ox.ac.uk/archive/z.html`, 1994 onwards.