Industrial Maintenance Modelled in SOCCA: an Experience Report

Tineke de Bunje[1] Gregor Engels[2] Luuk Groenewegen[2] Aart Matsinger[1] Mark Rijnbeek[2]

[1]Philips Research Laboratories
Prof. Holstlaan 4
NL-5656 AA Eindhoven
The Netherlands
e-mail: {bunje,matsingr}@natlab.research.philips.com

[2]Leiden University, Computer Sc. Dept.
P.O. Box 9512
NL-2300 RA Leiden
The Netherlands
e-mail:{engels,luuk}@wi.leidenuniv.nl

Abstract

A large industrial process, software maintenance, has been modelled by using the process modelling language SOCCA. The paper reports about the experiences with this trial. In particular, feasibility, expressiveness, quality, and the overall benefits of a formal SOCCA model are discussed and compared to the formerly existing informal process description. In order to illustrate the results, a well chosen process model fragment from the larger model is outlined in detail. It addresses in particular the human-intensive cooperation within the process and shows the seamless combination of technical components and human agent components in the SOCCA model. The main conclusions from this trial are that formal SOCCA models are suited to model realistic industrial processes and that due to an intrinsic modular structure of a SOCCA model even huge models remain reasonably readable and understandable.

1. Introduction

Research into software process technology (SPT) has made great progress during the last decade. On one hand, software process modelling concepts and languages have been investigated intensively. They range from activity-oriented to goal-oriented approaches, from high-level, abstract modelling languages to low-level, programming language like approaches. They focus on certain aspects of a software process such as synchronization of activities or try to offer comprehensive approaches to model all aspects of a software process in an integrated way. On the other hand, tool support for modelling activities as well as process-centred software development environments have been developed.

An overview of all this is given by the proceedings of the European as well as International workshop series on SPT as well as by the proceedings of previous ICSP conferences (e.g. [21, 17, 22]). In addition, the programme of any major conference in the field of software engineering comprises at least one session on SPT. Furthermore, research consortia have been funded like the ESPRIT basic research working group PROMOTER to bundle up research activities [9].

While the research agenda on SPT still contains a long list of open questions, researchers in the field are convinced that the results reached so far are mature enough to be tested and employed in realistic industrial situations. But, since great success stories on employing software process technology in industrial contexts are still missing, industrial software process improvement departments are still hesitating to apply research results in industry. Nevertheless, industry is keen to learn what the benefits of applying software process technology might be. Therefore, the software industry is running trial applications on transferring software process technology to industrial software development processes.

This article reports about such a trial, that was performed at Philips Research in close cooperation with the Software Engineering and Information Systems (SEIS) group at Leiden University. The basic idea was to set up a formal model for a real-life industrial process with the process modelling language SOCCA (Specification of Coordinated and Cooperative Activities). This language has been developed by members of the SEIS group during the last four years. While having been tested in several small-size case studies before, the case study described here is the largest and most complex one using SOCCA. This case study was mainly done during a master thesis project as reported in [18].

During this case study, the objectives of the industrial partner, Philips, and of the research partner, the SEIS group at Leiden University were to gain more insight in the following points:

1. feasibility: the possibility to formulate a SOCCA model for a large-scale industrial process;
2. expressiveness: the details of a SOCCA model not only in relation to the software development process, but also in relation to the embedding thereof in the more general business process;
3. quality: the quality of the SOCCA model compared to

the original, less formal process description;

4. metaprocess: the SOCCA modelling activity itself, also in relation to the field of application;
5. benefits: the surplus value of the SOCCA model for the (industrial) organization.

To this aim the paper has been organised as follows. Section 2 describes a concrete software process, i.e., a software maintenance process, as well as the informal model that Philips developed. Section 3 presents the SOCCA model for the maintenance process. Evaluation of the SOCCA modelling with respect to the above items in particular, is carried out in Section 4. Section 5 gives a comparison with related work. Conclusions and future research are the topics of the final Section 6.

2. The Case: A Maintenance Process

This section contains a short description of the industrial process to be modelled, the so-called SPI maintenance process, being defined and enacted at Philips Research. As the SPI environment consists of a couple of tools that are being modified and released independently of each other, a well-planned maintenance process was required. This maintenance process has been described in terms of a so-called <u>S</u>oftware <u>C</u>onfiguration <u>M</u>anagement <u>P</u>lan (SCMP). The SCMP description consists of four parts:

- the various categories of documents, each with its own lifecycle;
- the project phases; they constitute the maintenance lifecycle;
- the organization; it establishes the various roles of involved team members; these roles are developer, configuration manager, project leader, quality assurance manager, acceptance tester, configuration control board and event handler;
- the events, which here have the specific meaning of a change request or a problem report with respect to the SPI software to be maintained; each event lives through its own lifecycle, from “new” either to “rejected” or to “solved” and finally “closed”.

As a complete discussion of this maintenance process is far too page consuming for this paper, we restrict ourselves to the change request and problem report events and their lifecycles. Readers familiar with maintenance in practice, can roughly guess from the ingredients mentioned above, how the whole maintenance process has been organized.

Figure 1 is part of the SCMP. It presents the lifecycle for each event. Each event starts its life with status “new”, the top of Figure 1. Such an event, together with its status history, is stored and updated in the database called EventBase by a person responsible for this, the so-called SPI event

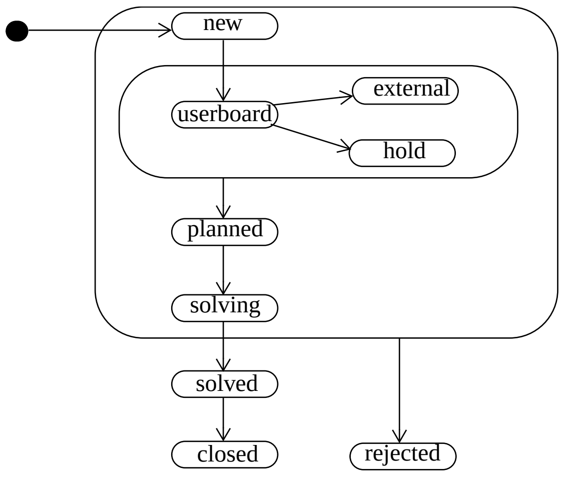

Figure 1. Lifecycle of an SPI event, in statechart notation.

handler. Normally the event will get the status “userboard”, indicating that it is to be discussed by a qualified group of user representatives. After that discussion the event will normally get the status “planned”, possibly after having asked for external advise - status “external” - or after having got a low priority - status “hold”. Status “planned” means that it has been decided in which release of the SPI environment the solution for the event will be incorporated. Status “solving” means that work on the solution and the implementation is continuing. Status “solved” means that this work has been successfully terminated, but still quite recently. Status “closed” means that the release in which the solution has been implemented, has been in use for two months or longer. In every status before “solved” it is possible that the request is considered as invalid for whatever reason. The event’s status is then changed into “rejected”, irrevocably.

From the above model fragment one can get a good impression of the informality of the SCMP, as being currently used. While the possible states of an event are defined, it is not fixed, which person (or better role) is allowed to trigger a state change of an event. This means that a precise description of the cooperation between the different roles, which are involved in the maintenance process, is missing. For instance, it is the SPI Event Handler that applies the status changes, but to that aim (s)he has to cooperate with the user board for one change, and with the developer for another. It is actually left to the SCMP reader’s imagination how this cooperation is to happen.

As the informality of this description was also evident to the designers at Philips Research of the SPI maintenance process, they were interested in possible answers to the following questions. Can a more formal process modelling language (PML) describe this process (feasibility)? Can such a description express more details? If so, which details, and how easily can one express them (expressiveness)? What is the quality of this more formal description (quality)? How does one arrive at such a description (metaprocess)? What are the advantages of such an approach for Philips (benefits)?

More directly related to the above process fragment, these open questions can be put forward much more concrete. Can a more formal PML describe the state change of an event from “planned” to “solving”? Which other component(s), with behaviour, is (are) responsible for this change (feasibility)? Can a more formal PML description express that a certain behavioural influencing exists between such a component and event? If so, can it express in detail how and when this cooperation occurs (expressiveness)? Is the composite description of an event, of the other components and of their cooperation readable, understandable, small, technical (quality)? How does one arrive at such a description of event and the other components, and of their cooperation (metaprocess)? What are the advantages of such a detailed description for Philips, not only for the process (model) designers, but possibly also for the process model users, i.e. those involved in the execution of the process while being directed by the enacted process model (benefits)? It is important to be aware, that at the time this research started, these questions - the general ones as well as the concrete ones - were open problems to the process designers at Philips.

This paper reports our findings in applying SOCCA to model the SPI maintenance. In so doing answers to the above questions will be given and discussed. For the sake of brevity, this paper restricts itself to the events, the SPI event handler and their cooperation. But this will already give a good impression how cooperation is precisely described in a SOCCA model.

3. The SOCCA model

The objective of this section is first, to give a quick overview of SOCCA, and second, to introduce the SOCCA model for SPI maintenance. By studying the latter, adequate experience in the look and feel of SOCCA will be acquired.

3.1. SOCCA

SOCCA is a graphical, object-oriented, high-level specification formalism together with a method for the analysis and design, and ultimately also the enaction of (software) process models, see e.g. [7, 18]. SOCCA is related to OMT [20], which it extends considerably by addressing communication more precisely. The name SOCCA is an acronym of Specification Of Coordinated and Cooperative Activities.

Like OMT, SOCCA is an eclectic, see [12], process modelling language. As such it consists of several sublanguages carefully integrated with each other. These sublanguages are Class Diagrams (CDs), State Transition Diagrams (STDs) and Paradigm. The name Paradigm is itself an abbreviation. It means PARallelism, its Analysis, Design and Implementation by a General Method [23].

Globally speaking, building a SOCCA model consists of several steps. In the first step the static structure of the problem and its solution is specified. Like in OMT this is done by means of a CD. This CD determines the relevant object classes, their attributes, operations - or methods - and relationships, among which the well known is-a relationship and part-of relationship, and furthermore the so-called uses relationship. This uses relationship - or import-export relationship - specifies in which classes the various (export) operations can be imported, i.e. can be called.

In the second step the external or visible behaviour of each class is specified by means of an STD. All export operations of such a class may appear as a label of a transition of this STD. In this STD unlabelled or differently labelled transitions can also occur. A transition labelled with an export operation specifies the external, visible state change of this STD resulting from a call somewhere to the operation this transition is labelled with. In this way it is made visible that this particular call is taken care of. An unlabelled or differently labelled transition specifies the external, visible state change of the external STD resulting from internal, invisible behaviour, hidden inside the same object instance of that class. By means of the external STD all possible execution sequences of its export operations are specified. As these executions can be simultaneous, the sequences only specify the order of the beginnings of these executions, and not the order of the executions. These sequences can be interlaced with global effects resulting from hidden internal behaviour. This specification of the external behaviour is comparable to the “dynamic model” in the OMT approach.

The third step in the SOCCA approach consists of specifying the internal behaviour of each export operation by means of a separate STD. Such an STD usually has a state denoted as “operation not active” or something similar, and a transition leaving this state and labelled with “act-operation”, which means activate that operation. Other transitions of an internal STD can be labelled with “call other operation”; only “other operations” are allowed which are imported into this class according to the uses relationship. Transitions may also remain unlabelled, or labelled with a short phrase referring to the intuitive meaning of that particular step in the internal behaviour.

The idea behind this organization of the model is as follows. The actual external behaviour of each object is described by the external STD of the corresponding class.

This shows which sequence of export operation calls is actually being executed. The calls to these export operations are part of the hidden behaviour within the various internal STDs. The execution model of this huge number of STDs is, that each STD is being executed on its private processor, parallel to all the other STDs. This means that the communication between these STDs still has to be specified, as these parallel behaviours clearly depend on each other.

Therefore, the fourth step of the SOCCA approach consists of specifying the coordination and cooperation between these behaviours. This is done by means of Paradigm, see e.g. [23] for a more extensive introduction. Paradigm specifies communication between STDs by using the notions of manager (process), employee (process), subprocess and trap.

The global idea is as follows. An STD is either a manager, or an employee (of a manager), or both but not an employee of itself. Each communication that takes place between STDs, is between a manager and several employees. To that aim each employee is divided into subprocesses, representing restrictions of the original STD and having the meaning of temporary behaviour restriction of that STD. The manager, by being in a state, prescribes the current subprocess to the employee, meaning for the employee to restrict its behaviour according to that subprocess. To that aim the states of the manager are labelled with the subprocesses it is supposed to prescribe there. On the other hand, each subprocess has one or more traps, representing a part of the states of that subprocess that once entered cannot be left as long as that subprocess is being prescribed. By entering a trap within a subprocess, the employee allows its manager to make a transition to a state where a next subprocess is being prescribed. So entering a trap is the indication that the current subprocess prescription can be changed into the next one, as far as the employee is concerned. To that aim the transitions of the manager are additionally labelled with the trap that allows that transition.

In SOCCA Paradigm is used as follows. Each external STD is manager process of each internal STD belonging to the same object (class instance). In addition each external STD is manager process of each internal STD belonging to any object that contains a call to an operation exported by the object this external STD belongs to and imported in the object this internal STD belongs to. So the employee processes are the internal STDs. Each internal STD is employee of at least one manager, the external STD of the object it belongs to. If this internal STD contains calls to export operations, it also has those external STDs as managers where such an operation occurs as transition label.

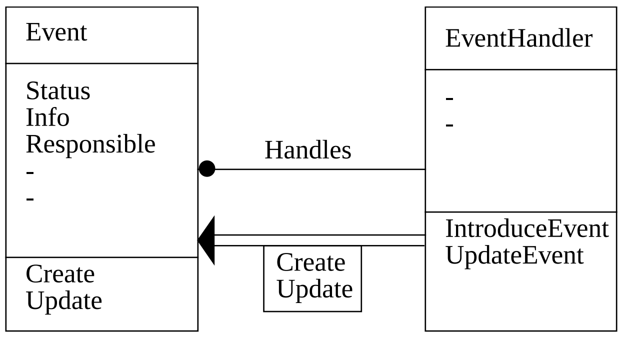

Figure 2. Class Diagram with Uses Relationship for the Process Fragment concerning Event.

This concludes our general discussion of SOCCA's modelling steps. This short and abstract introduction into SOCCA will become more clear in the next section.

3.2. A fragment of a SOCCA model for SPI maintenance

The SOCCA model for SPI maintenance as discussed in [18] describes all aspects of the maintenance process. In particular, it pays a lot of attention to the so-called Fagan inspection and corresponding meeting in relation to the documents produced by the maintenance process. In this way the suitability of SOCCA for specifying typical human cooperation has been investigated and demonstrated. For our purpose here the details of the various human roles in the inspection and meeting are too numerous. Therefore we concentrate on just one very restricted process fragment, the events, which are handled by one human agent, the event handler. With respect to this fragment we will mainly follow [18], but we leave out all details concerning the event handler being a specialization of the organization's employee.

3.2.1. SOCCA's First Step: Static Structure

According to the first SOCCA step, a class diagram concerning events has to be developed. Figure 2 presents this class diagram. Only two classes have been drawn, Event and EventHandler. Event contains all relevant information concerning a particular event. To that aim it has attributes for data like: its current status, its current phrasing, the person(s) responsible for the phase leading to the next status. Moreover, Event offers two export operations, Create and Update(). As these two operations are exported exclusively to EventHandler and as Event itself does not import any operation, EventHandler is the only class that is of immediate interest to Event.

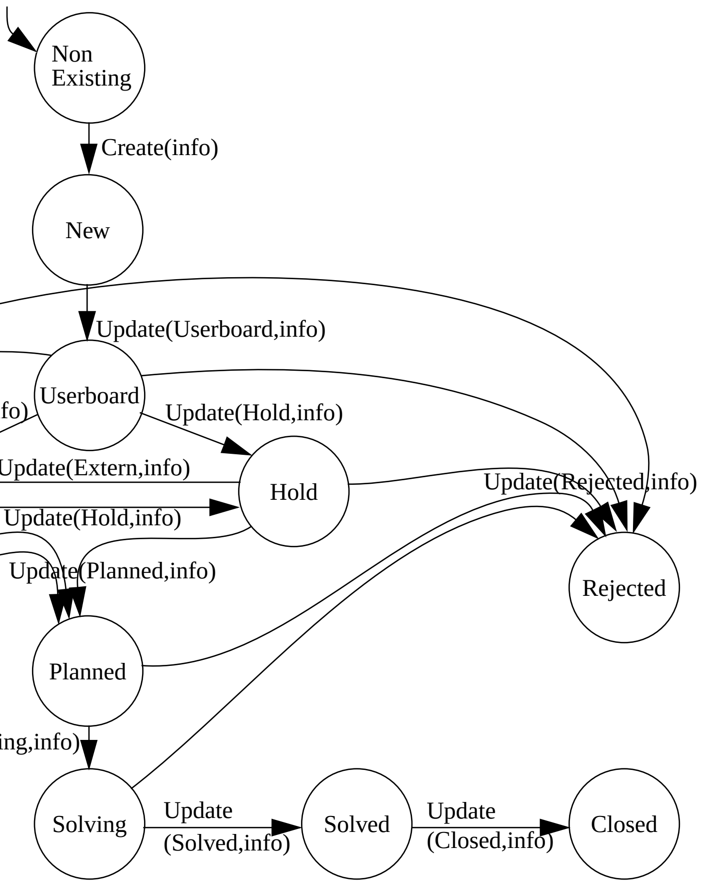

Figure 3. External behaviour of Event: organizational view, folded out

The graphical symbols in the figure have the following meaning. A three-layer rectangle denotes a class. The class name is indicated in the top layer. The middle layer gives the attributes, and the bottom layer gives the export operations or methods. Single lines between classes represent general relationships, their names indicated by the accompanying label. A black dot at the end of such a line represents a multiplicity greater than or equal to 0, whereas no dot at all at the end of a line represents a multiplicity of exactly 1. Thus Handles represents a 1 : n relationship between exactly one EventHandler object and an arbitrary number of Event objects. A double line with a black arrow head and labelled with a small rectangle containing one or more export operations, represents a uses relationship, or import-export relationship, the arrow pointing towards the class providing the operations to be exported. Thus EventHandler imports the methods Create and Update from Event. This finishes the first step of the SOCCA approach for this process fragment. By now the static structure is fixed.

3.2.2. SOCCA's Second Step: External Behaviour

The second step starts modelling the dynamic structure, by defining for each class the external or visible behaviour as consisting of the allowed calling sequences of the methods provided by that class.

The external behaviour of Event is here presented in different forms, called **views**, each one a different STD.The first one, see Figure 3, comes closest to the statechart from Figure 1. The update operations with a different value as status parameter correspond to ever so many different transitions. This view of the external behaviour is called **folded out**, in contrast with the more concise representation from Figure 4, having only one transition labelled by Update.

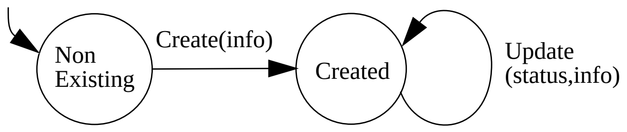

Figure 4. External behaviour of Event: folded organizational view

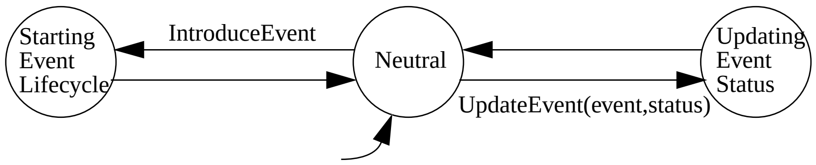

Figure 5. External behaviour of EventHandler

This STD is a **folded** view. Later on, Figure 8, some unlabelled transitions will be added to this STD, reflecting some relevant communication details. Without these details the STDs are called **organizational**, representing a global management view of the external behaviour. With these details, as in Figure 8, such an STD is called **communicative**.

The advantage of the folded out view is the clearer visualization of the calling sequences. For instance, from the STD in Figure 3 it is immediately clear that any alternation in transition between state Extern and state Hold is allowed, and in addition, from state New no transition to state Rejected is permitted. Both facts make the STD behaviour deviate from the state chart behaviour. This is not visible from the STD in Figure 4.

The advantage of the folded view is in being more concise. For our discussion we prefer the folded view. In both Figures 3 and 4 the starting state is Non Existing. First Create has to be called. After that Update has to be called. In the folded out view of Figure 3 it is moreover specified how many update calls are permitted, with which values for its status parameter, and in which order these values should appear. In the folded view of Figure 4 this remains unspecified. In this representation the external behaviour STD abstracts from the different states and keeps the current status only as an attribute value of each instance of class Event. It will be part of the communication description of the SOCCA model to specify in which sequence and in

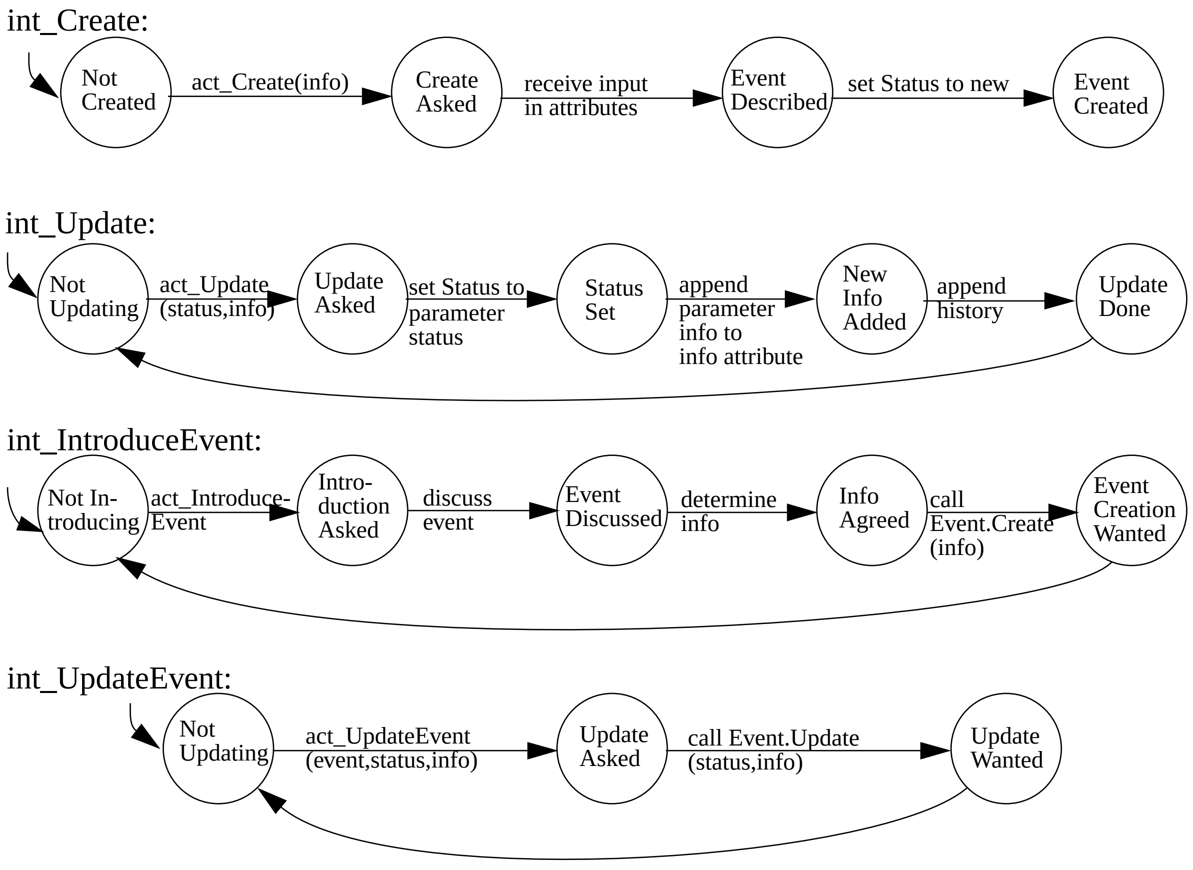

Figure 6. Internal Behaviours: int_Create, int_Update, int_IntroduceEvent and int_UpdateEvent

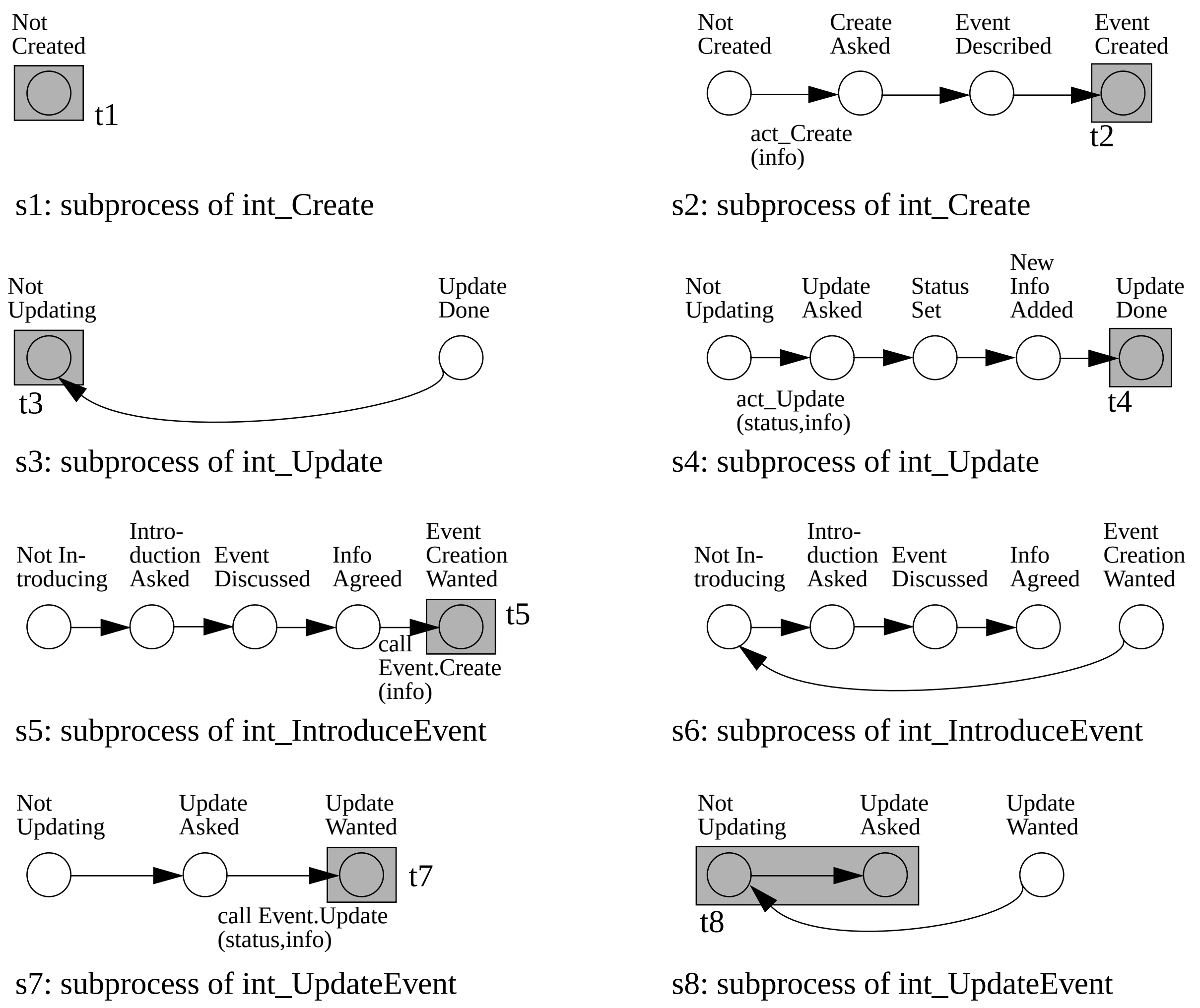

Figure 7. Subprocesses and traps of Event's employees

which situation the status of an event may be changed by a call to the Update operation.

The two export operations of Event - Create and Update - are called from the internal behaviours of EventHandler, as it is reflected by the uses relationship in Figure 2. Before discussing these details, we present the external behaviour of EventHandler in Figure 5.

The external behaviour of EventHandler is a composition of its own operations. The starting state is Neutral. Note that no sequencing whatever of operation calls is enforced by this external behaviour. The EventHandler, upon starting to react on a call, returns to its neutral state. From that state any new call can be reacted on.

3.2.3. SOCCA's Third Step: Internal Behaviour

The third step of our modelling brings us to the STDs of the various internal behaviours, one for each method. In Figure 6 the internal behaviours of Create, Update, IntroduceEvent and UpdateEvent have been visualized, referred to as int_Create, int_Update, int_IntroduceEvent and int_UpdateEvent respectively. The corresponding starting states are Not Created, Not Updating, Not Introducing and Not Updating. Note the calls of Create and Update inside the int_IntroduceEvent and int_UpdateEvent respectively. This is conform the uses relationship as given in Figure 2. Apart from the labels with prefix act_ and those with call, the other labels are just an informal reference to their meaning.

It is crucial to keep the execution model of these numerous STDs in mind: each STD is supposed to execute on its own, private processor. So all STDs are simultaneously active. In order to handle behaviour that influences other STDs, cooperation between these STDs takes place in a coordinated manner.

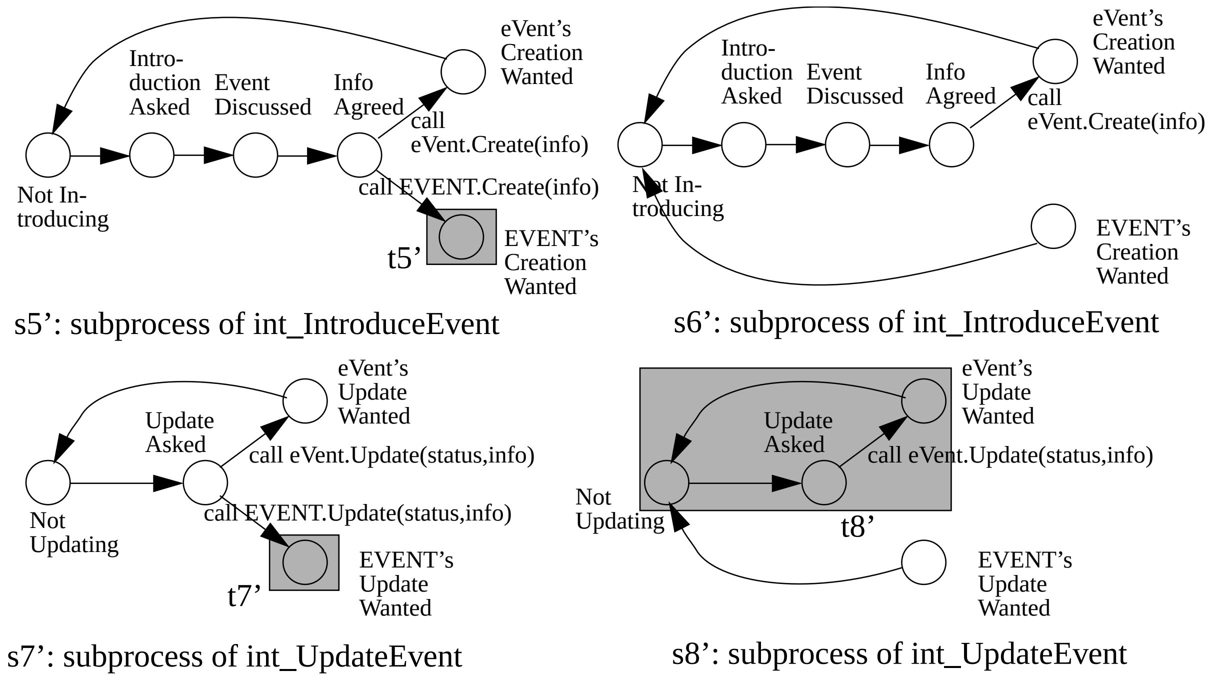

Figure 8. Instance-oriented subprocesses and traps of Event's (EVENT) calling employees

3.2.4. SOCCA's Fourth Step: Communication

Step four of the SOCCA modelling addresses the communication between different objects by specifying how they cooperate. This is done by means of Paradigm, as explained e.g. in [7]. The managers then coordinate the cooperation with and between their employees. In SOCCA the managers are the external behaviours. So the external STD of Event is a manager. Its employees are the internal behaviours of Create and Update, being the internal behaviours belonging to Event itself, and moreover the internal behaviours of IntroduceEvent and UpdateEvent of an EventHandler, being the internal behaviours containing a call to Create or Update. The temporary behaviour restrictions of the employees as resulting from the coordination, are modelled as subprocesses or subSTDs. A trap, a part of the subprocess' state space that cannot be left as long as the current subprocess is being executed, is used to express the commitment that a relevant part of the current subprocess has been executed and that the employee will not revoke it during this subprocess execution.

In our SPI maintenance example the employees of Event have subprocesses and traps as presented in Figure 7. Subprocesses are visualized as subSTDs of the original employee, and traps are visualized as shaded rectangular areas around the states belonging to that trap. For the sake of brevity only, most labels of transition have not been repeated. Note that subprocess s6 does not have a trap. This is because after s6 has been prescribed to Event, resulting from a call to Event.Create, there is no need for a second call to Create of the same Event instance. Therefore, with respect to the same Event instance subprocess s5 is not needed after s6, so no trap is needed. The graphical notation from Figure 7 links on to the type-like descriptions of behaviour and data from Figures 2, ..., 6. But as we have observed, communication has a strongly instance-oriented character: the calling of Event.Create within the operation IntroduceEvent of EventHandler is towards one particular Event instance and not towards other Event instances.

In view of the instance-oriented character, subprocesses s5, ..., s8 are revisualized in a more instantiated manner in Figure 8. Here the particular Event instance is denoted as EVENT, whereas the other instances are jointly referred to as eVent. Note how s6' allows for calling Create of other Event's instances after starting EVENT's creation.

The names of the subprocesses and traps, indicated in Figure 7 as s1, ..., s4 and t1, ..., t4, and in Figure 8 as s5', ..., s8' and t5', t7', t8' respectively, will be used in the description of the manager's coordinating role. As said before, it is Event - or more precisely, EVENT on the instance level - that plays this manager role. Paradigm organizes this as follows.

First of all, the organizational view of the external behaviour is taken as starting description of the manager process. But it is expanded into a communicative view by replacing each transition labelled with an operation by two

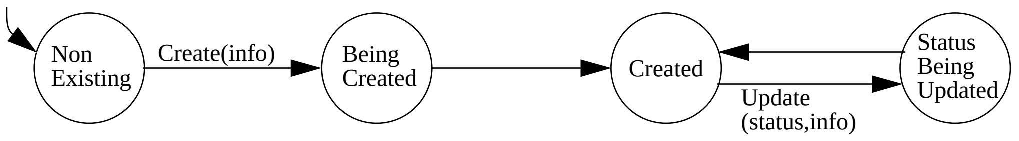

Figure 9. EVENT’s external behaviour: folded, communicative view

subsequent transitions, the first one still labelled with the same operation, and the second one unlabelled. The labelled transition represents starting the operation’s execution because of a call. The extra, intermediate state in between these two transitions represents: the call of the operation has resulted in starting the operation’s execution. So this execution has not necessarily been finished as yet. The unlabelled transition represents informing employee processes that the execution has proceeded far enough to handle a possible next call. (In some cases, not in this example, the unlabelled transitions can be omitted: having started an execution then also means the readiness for starting another.) The communicative view of EVENT’s external behaviour is presented in Figure 9. Compare this with Figure 4. (The folded out, communicative view can be easily constructed from Figure 3.)

Each of EVENT’s states (from the communicative view) is labelled with one or more subprocesses for each of EVENT’s employees. EVENT, at each time instant being in exactly one of its states, prescribes these subprocesses to its employees as their current subprocess (or their current behaviour restriction) - for each employee one subprocess. Moreover, each of EVENT’s transitions (from the communicative view) is labelled with one trap for each of its employees. Such a trap is trap of a subprocess prescribed in EVENT’s state where the transition is outgoing; and the states of that trap are also part of the subprocess prescribed for the same employee in EVENT’s state where the transition is incoming. It is the employee, by entering this trap of its current subprocess, that allows this transition. The transition can be made only after each employee has allowed it.

The labelling of EVENT’s state and transitions with subprocesses and traps is given in Figure 10. The label order for the employees, also indicated in the figure, is: int_Create, int_Update, int_IntroduceEvent, int_UpdateEvent.

More intuitively, the above modelled coordinated behaviour between a human EventHandler and an instance of Event is as follows:

In the beginning, EVENT is in state NonExisting, and the four employees are in the subprocesses s1, s3, s5’, and s7’, respectively. EVENT waits for a call of Create, which is invoked during the execution of IntroduceEvent within subprocess s5’, when the EventHandler reaches trap t5’ for this EVENT. Subprocess s1 and trap t1 indicate that EVENT is ready to be created. At the same time, Update can not be started, as the corresponding subprocess s3 can not leave trap t3. While it is possible that already an Update for this EVENT has been invoked within subprocess s7’, this subprocess can not continue as it is caught in trap t7’.

After Create has been called (t5’ has been entered; t1 was already entered), int_Create is started (s2 is being prescribed), and int_IntroduceEvent may continue after the call (s6’ is being prescribed). As soon as EVENT has been created (t2 has been entered) updates of EVENT’s status may be handled. (Note how the small size of one state of t2 establishes the sequentialization of these steps.) In view of the status changes, it is checked whether such an update is

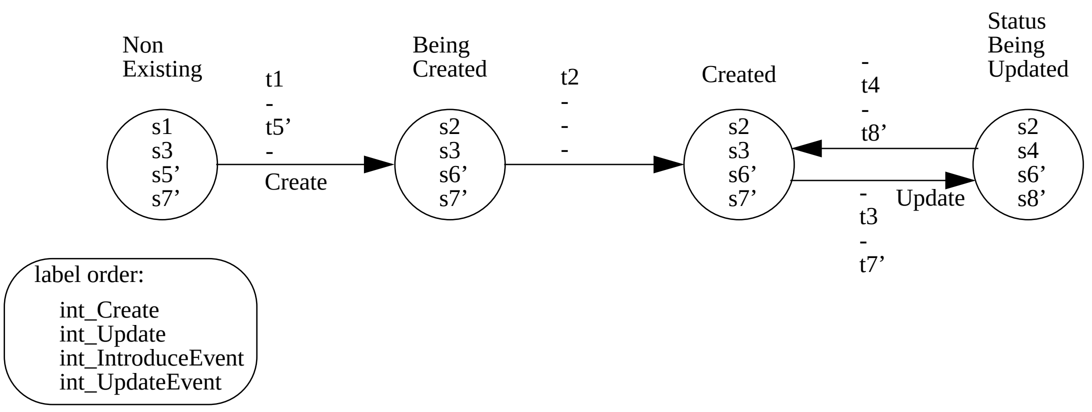

Figure 10. EVENT’s external behaviour as manager process: folded communicative view

wanted (trap t7' has been entered), and whether the update can be handled (trap t3 has been entered). If so, the handling starts (s4 is being prescribed), and int_UpdateEvent is allowed to proceed after the calling, but a new call for updating the same EVENT is not yet allowed (s8' is being prescribed). After the update has been finished (t4 has been entered), and after int_UpdateEvent is no longer asking for this update (t8' has been entered), the old situation where updates may be requested, is being restored (s3 and s7' are being prescribed).

The reader should note how the differences in trap size can be used to model different types of coordination: the larger a trap, the more asynchronous as well as parallel the corresponding part of the communication. For instance, t1 and t5' are single state traps, so nothing happens to the corresponding STDs in between entering the trap and the next subprocess prescription. This is synchronous communication. As another example, the small size of trap t2 lies at the basis of the complete execution of s2 before any update request to the same Event instance - EVENT - can be considered. This is sequentialization.

A new call of an Update request to the same EVENT parallel to the treatment of the former Update request could be allowed by enlarging the trap t4 of subprocess s4 to all states of s4. This would be an example of parallelization.

This concludes the description of a very small part of the complete SOCCA model for SPI maintenance. It shows how the cooperation between a human (EventHandler) and a non-human (Event) agent can be described precisely. In the same way, the cooperation between the EventHandler and all other human agents (roles) involved in the maintenance process, like a developer or a user representative, has been modelled (cf. [18]). In comparison to the SCMP, the local description of possible states of an event (cf. Figure 1) is distributed over the behaviour and communication description of all involved human agents. From the SOCCA model one can conclude, exactly when and also by which class instance the methods are to be called, and whether this communication is synchronous or asynchronous (and what is the degree of asynchronism).

4. Evaluation of the SOCCA model

Although the above model only represents a very small part of the complete SOCCA model for SPI maintenance, the reader by now can have a good overall impression of SOCCA's modelling power. For the process designers at Philips the experience with the SOCCA model and the SOCCA modelling process made it possible to answer their open questions from Section 2 as follows.

Feasibility: the SOCCA model indeed describes the behaviour of Event as well as of EventHandler, see Figures 3 (or 4 or 9) and 5. More concrete, Event's behaviour description contains the change from “planned” to “solving”, as was one of the questions in section 2. In the folded out view this is immediately visible, see Figure 3. In the folded view this can be derived from analysing the order of the parameter values of status corresponding to the transitions labelled with the operation Update in Figure 4 or in Figure 9. From Figure 2, the uses relationship concerning Update, it follows that EventHandler is responsible for this (and other) change(s) of Event. (From the complete overview of uses relationships, see Figure 2, it moreover follows that EventHandler is the only responsible.)

Expressiveness: the SOCCA model presents two levels of expressing how and when the cooperation between Event and EventHandler occurs. Through the internal behaviour of int_UpdateEvent, see Figure 6, EventHandler's role as event updater is described. Here the only transition occurs corresponding to calling Event's operation Update. So this role describes on a global level the “how and when” of the cooperation. “How”: by performing the call, and “when”: after EventHandler's operation UpdateEvent has been started with “solving” as the correct value of status. (From the complete model it follows that only Developer can start this, in the role of software producer.) On detailed level the “how and when” is precisely specified through the subprocess s3, s4, s7' and s8', the traps t3, t4, t7' and t8' and EVENT as manager process, see Figures 7, 8 and 10.

Quality: as feasibility and expressiveness are being addressed separately, we here concentrate on the general accessibility of the SOCCA model. The richness in expressiveness of SOCCA has its price with respect to readability, understandability, size and technical level. The above model fragment is itself very small; 2 classes with 2 operations each, and 1 uses relationship involving 2 of these operations. This then leads to 6 STDs (apart from the different view possible), 8 subprocesses, 7 traps, 1 manager and 4 employees. Here the size is still small, and although the technical details of the Paradigm part in particular are not simple, the fragment remains reasonably readable and understandable.

However, the model as a whole is not just large but huge. To give an impression of the size, the model in [18] consists of 13 classes (apart from 7 others that are aggregations or generalizations/specializations of these) and 43 operations (17 of which are copies of others, being inherited). This then results in 56 STDs, 39 of which are different. For the communication another 138 subprocess and 150 traps were defined for the 26 different internal STDs. That means 26 different employees are to be controlled by 13 managers. This certainly disqualifies the model as easily readable. Because of the many technical details it is moreover not an easy task to understand the model. On the other hand, the model itself has different levels of expressiveness. The lowest level is the class diagram together with the uses relationships, cf. Figure 2. Even for the complete model this is easily readable as well as understandable, even though some 13 (or 20) classes are involved with 14 uses relationships. The next level of expressiveness consists of all STDs, i.e. all external and internal behaviours without the Paradigm details, cf. e.g. Figures 4, 5 and 6 (restricted to one view). For the complete model this level of expressiveness consists of 56 STDs. This is certainly large, but still reasonably readable and straightforwardly understandable, mainly because of the structuring effect of SOCCA's object-orientation. On this level the "how and when" of the cooperation already are globally specified.

Metaprocess: the steps to be taken in the SOCCA modelling process are very clear. For the above fragment they consist of the steps taken in the subsections 3.1, ..., 3.4. The eclecticism of SOCCA requires a detailed integration of its various sublanguages, such that quite a number of consistency constraints have to be fulfilled. On the one hand, this gives structure to the next steps of the metaprocess: the occurrence of call Event.Update in an internal behaviour of EventHandler requires the occurrence of Update in the uses relationship of Figure 2. On the other hand, this also gives guidance to possible iteration of earlier steps: Event's prescription of subprocess s8' (to EventHandler's internal behaviour int_UpdateEvent) in alternation with the prescription of s7' leads to a revision of EventHandler's external behaviour in Figure 9, the communicative view instead of the organizational view.

Benefits: the SOCCA model first of all entails positive answers to the questions of feasibility, expressiveness and metaprocess, and a sufficiently satisfying answer to the question of quality. The much clearer process description as compared to the original one, brought the process designers at Philips a greater confidence in the model. The new model indeed describes the actual process more accurately. This greater confidence is based on facts like the following.

The above model fragment explicitly says, only EventHandler can update Event's status. The original description is vague on this point. The complete SOCCA model says, only Developer can ask EventHandler to update Event's status from "planned" to "solving". The original description leaves this open.

In particular, the description of the human-intensive cooperation was considered as clarifying in being very detailed as well as allowing many degrees of asynchronism in the communication. It is apparent from the above example too: EventHandler's behaviour and communication in SOCCA are very precise, in particular with respect to (a)synchronism. In the original description these points were left to the reader's imagination.

SOCCA moreover provides process guidance as easy as process enforcement. So it is very suited for modelling different kinds of human behaviour and human interaction. In the above model fragment the guidance is expressed in the external behaviour of EventHandler. There is no strict sequence in the calls to its export operations. Even after the EventHandler has started some or all internal behaviours, it is the freedom of the execution mechanism in which order - strict, interleaved, or even really parallel - these internal behaviours proceed. The internal behaviours of one object modelling a human agent, can be compared to the workload indicated within the workspace for that human agent: (s)he can choose with which internal behaviour - to be interpreted as a (small, individual) task or role - to proceed and for how long.

In the above case the validation of the model took place through inspection by various persons at Philips involved in the process. The experiments mentioned moreover show that animation and simulation, based on suitable metrication, is within reach. This would be of great help towards validation of the model, by comparing simulation results with the real process. The actual benefit of the validation by inspection was and is a greater confidence in the process as described by this more precise model. This growth in confidence occurred at two partners. First of all at the designers (at Philips) of the maintenance process as they preferred this more precise description. But also at the users (at Philips) of it, as they see better possibilities for metrication and simulation based on the new model.

5. Related work

Within the world of software process modelling (SPM), there are many different SPM languages. Quite a lot of these are concentrating on formulating the model for the software process directly in an executable language, e.g. EPOS [6], Adèle-Tempo [3], ALF [5], Marvel [2], Appl/A [14]. Other approaches offer diagrammatic support for formulating such a model on a higher, more abstract level first, before translating this into some executable language, e.g. LEU [11], SPADE [1], Process Weaver [8]. Also approaches from the first category are currently being upgraded towards the second category, e.g. Merlin by extending it with Escape [16]. Sometimes a diagrammatic support is given not for the first formulation, but for later evaluation, e.g. in PADM [5] so-called RADs - Role Activity Diagrams - are used after the first, high-level BM (Basic Model) specification.

SOCCA belongs to the second category, like SPADE and LEU, although at the moment SOCCA does not offer any implementation of the models formulated. Currently work is in progress to combine SOCCA models with simulation, and after that with enacting execution. On the other hand, comparing the graphical approaches in SOCCA with the Petri nets in LEU, SPADE and Process Weaver, SOCCA offers clearer modularization by means of its classes and their STDs. Also in the communication, SOCCA offers the full range from synchronous to asynchronous, allowing even for gradations of asynchronism, depending on the size of traps. In the original Petri net approach there is only synchronous communication. So every form of asynchronous communication is to be "simulated" through a synchronous form.

A comparison with Escape-Merlin shows that both start with a class diagram. In Escape statecharts are used for the behaviour specifications. So with respect to modularization through the various STDs SOCCA and Escape are more or less similar. Although it is not mentioned in [16], there are possibilities for discriminating between visible and hidden behaviour, see [13]. But also here the details of the communication specification in SOCCA are different from those in statecharts, which only offer synchronous communication.

What essentially makes SOCCA's communication specification so different from those in Petri nets and in statecharts, is the notion of subprocess. This notion describes a whole part of the behaviour being permitted after some message has been received, instead of just one step of the behaviour being permitted. Thus a graphical representation of a subprocess reflects not only the immediate consequence of a communication but also the more long term consequences. This turns out to be of great help in modelling real life processes such as SPI maintenance.

Apart from a comparison to SPM languages, SOCCA can also be compared to OMT [20]. In class diagram and in external behaviour, the similarity is striking. Also here the differences come in with the communication description. In that respect OMT is known to be not very detailed. On the other hand, the object flow description in SOCCA is not yet graphically supported, so the effect of the behaviour and the communication for the attributes from the class diagram still has to be graphically expressed. In OMT behavioural and communicative effect on the attributes is described in stages through several diagrams.

6. Conclusions and Further Research

The concrete benefits have been discussed above. On a more general level the benefits consist of bringing the following possibilities within reach. The clearer description of human-intensive cooperation leads to more to the point discussions about model variations. Not only choices between guidance and enforcement, and between parallelism and sequentialization can be clearly made. Also the embedding of the model into a larger business context can be incorporated. Furthermore, other process fragments, such as version-management, can be straightforwardly added.

By means of some suitably chosen value function, things like reward, profit, cost, duration, number of (certain) steps can be measured, completely similar to what is being done in operational research, e.g. in Markov decision processes or in PERT planning. (In particular, time is a crucial notion for a complete picture of a process. Duration, as mentioned here, then could be best defined as the sojourn time in a state, like in semi-Markov decision processes. This duration actually expresses how long it takes for a transition to occur. The transition itself is instantaneous. See e.g. [19].) Choosing a value function for a process model is also referred to as adding metrics to a process, or metrication of a process model. Metrication then facilitates the planning and coordination, such that in principle optimization with respect to what is measured, comes within reach. This also gives insight in which parts of the model are relevant for what is being measured. The better the model reflects the real process, the better the optimization with respect to the model's metrics corresponds to optimizing the actual ongoings of the real process. So developing a model that is both accurate (in reflecting the real process) and precise (in its formal and unambiguous formulation) is an essential step towards better management of the corresponding real life process.

Our main conclusion is that the surplus value of the SOCCA modelling approach is considerable. Precisely by being so rich and detailed with respect to behaviour and communication, SOCCA offers far reaching integration possibilities between technical, highly automated processes and human intensive business processes into one model. SOCCA's flexibility in metrication provides a sound basis for controlling the model, and through the model the real process.

From modelling SPI maintenance, and also from modelling other industrial processes, we learned that a SOCCA model for such a situation can be straightforwardly constructed. In general such a model is very detailed, particularly the communication specification within the model. This has the advantage that a specification is very precise, but the disadvantage that a specification is not just large, but huge. A first thing one can do, is to concentrate on the different levels of expressiveness as discussed above. A second possibility is the introduction of templates and communication patterns for the purpose of description reduction; see also [12, 10] for related thoughts. This is still a topic of further research.

During the modelling activity it was a handicap not to have a SOCCA environment. Such an environment certainly would have had the following advantages: figures can be drawn and adapted much quicker; many consistency checks can be done automatically; it can give support in determining various communication details (e.g. a call means entering a trap). The development of such an environment has been started, so we will report on that in the near future.

SOCCA's modelling approach, we have learned, is very useful for human-intensive communication. Not only the communication between a human agent and a computerized component, but also between human agents. So the complete organization around the problem situation can be modelled, too.

This makes it difficult to decide where to stop modelling, if the original question was to specify the problem situation accurately. As a positive result we have also learned that a SOCCA model is easily adaptable and extendable. This is especially true in view of metrication: to the aim of measuring whatever feature already present in the model, it turns out that registration can be straightforwardly added. If the model is lacking the feature to be measured, first the model is to be adapted in that respect, and then the registration can be added. Quite often both steps are straightforward, allowing for much reuse of the existing model fragments. With respect to the difficulty to decide where to stop modelling, ongoing research indicates the usefulness of metrication goals. Such goals provide a kind of step-wise refined guidance for the modelling itself, see [4]. In another future paper we will report our findings concerning metrication as a modelling guideline.

In view of adapting a SOCCA model towards metrication it is interesting to mention ongoing research on process evolution. It presents a formal specification of evolutionary process (model) change, using a special component called Wodan. A topic of further research is then to investigate how Wodan can be tailored towards adapting or extending a SOCCA model for metrical purposes only.

Acknowledgements

We thank Volker Gruhn for his very constructive remarks. This has led to a clearer and more concrete description of the evaluation part.

References

[1] S. Bandinelli, A. Fuggetta, C. Ghezzi, L. Lavazza: SPADE: an Environment for Software Process Analysis, Design and Enactment. In [9], 223-247.

[2] N. Barghouti, G. Kaiser: Scaling up Rule-based Development Environments. Intern. J. on Software Engineering and Knowledge Engineering, World Scientific, 2(1), 59-78, March 1992.

[3] N. Belkhatir, J. Estublier, W. Melo: Adèle-Tempo: an Environment to Support Process Modelling and Enaction. In [9], 187-222.

[4] T. de Bunje, G. Engels, L. Groenewegen, M. Heus, A. Matsinger: Towards Measurable Process Models. In C. Montagnero (ed.): Proc. of EWSPT'96. Springer, Berlin, LNCS, 1996.

[5] R. Bruynooghe, R. Greenwood, I. Robertson, J. Sa, B. Warboys: PADM: towards a Total Process Modelling System. In [9], 293-334.

[6] R. Conradi, M. Hagaseth, J.-O. Larsen, M. Nguyên, B. Munch, P. Westby, W. Zhu, M. Jaccheri, C. Liu: EPOS: Object-oriented Cooperative Process Modelling. In [9], 33-70.

[7] G. Engels, L. Groenewegen: SOCCA: Specifications of Coordinated and Cooperative Activities. In [9], 71-102.

[8] C. Fernström: Process Weaver: Adding Process Support to Unix. In [17], 12-26.

[9] A. Finkelstein, J. Kramer, B. Nuseibeh (eds.): Software Process Modelling and Technology. Research Studies Press, Taunton, UK, 1994.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Mass., 1995.

[11] G. Graw, V. Gruhn: Process Management In-the-many. In [22], 163-178.

[12] L. Groenewegen, G. Engels: Coordination by Behavioural Views and Communication Patterns. In [22], 189-192.

[13] D. Harel: Statecharts: a Visual Formalism for Complex Systems. Science of Computer Programming, 8, 1987, 231-274.

[14] D. Heimbigner, S. Sutton, L. Osterweil: Language Constructs for Managing Change in Process-centred Environments. In: Proc. of the 4th ACM/SIGSOFT Symposium on Software Developments Environments, December 1990. In ACM SIGPLAN Notices 15(6), 206-217.

[15] L. Hertzberger (ed.): Intelligent Autonomous Systems (Intern. Conf., Amsterdam, December 1986). Elsevier, Amsterdam, 1987.

[16] G. Junkermann: A Dedicated Process Design Language based on EER Models, Statecharts and Tables. Proc. of the 7th Int. Conf. on Software Engineering and Knowledge Engineering, Rockville, Maryland, 1995, Knowledge Systems Institute, 487-496.

[17] L. Osterweil (ed.): Proc. of the 2nd Intern. Conf. on the Software Process, Berlin, February 1993. IEEE Computer Society Press, Los Alamitos, Cal, 1993.

[18] M. Rijnbeek: Modelling a Software Process using SOCCA. Master Thesis, Leiden University, Computer Sc. Dept., Int. Rep. 95-05, 1995.

[19] S.M. Ross: Applied Probability Models with Optimization Applications. Holden-Day, San Francisco, 1970.

[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: Object-Oriented Modelling and Design. Prentice-Hall, Englewood Cliffs, 1991.

[21] W. Schäfer (ed.): Proceedings of the 8th International

Software Process Workshop. IEEE Computer Society Press, 1993.

[22] W. Schäfer (ed.): Software Process Technology (EWSPT’95, Noordwijkerhout, The Netherlands). Springer, Berlin, LNCS 913, 1995.

[23] M. van Steen, L. Groenewegen, G. Oosting: Parallel Control Processes: Modular Parallelism and Communication. In [15], 562-579.