Blackboard Systems in SOCCA

Process Evolution visualised by Reproductive, communicating Blackboard Systems

Carla Spruit

Augustus 1997

MASTER'S THESIS
Department of Computer Science
Leiden University
P.O.Box 9512
2300 RA Leiden
The Netherlands

Abstract

Goal of this thesis is to specify Blackboard Systems in SOCCA and to investigate the possibility to model evolving processes by means of communicating Blackboard Systems. In the field of Software Process Modelling, the ability to model evolution is of great interest as it can support the structured thinking and simulation of processes.

Blackboard Systems were originally developed in the field of artificial intelligence as a method to organise problem-solving programs. The problem-solving is dynamically controlled by automated experts that communicate with each other through a global database, called 'the Blackboard'.
The evolution of the problem-solving is visualised by the recording of the successive modifications on the Blackboard.
By replacing the automated experts by humans, Blackboard Systems naturally represent evolution of human collaboration processes.

First, a Blackboard System that is fit to represent human collaboration processes is designed by using SOCCA, a process modelling language that models automated and human parts of a system in exactly the same way. As no automated parts are included in the proposed Blackboard System, the SOCCA model will be a model of a completely non-automated process.
Like processes, this Blackboard System has to be able to create, influence and terminate other Blackboard Systems. They must be able to operate concurrently and communicate with each other.

Secondly, the representation of evolving processes by means of Blackboard Systems is illustrated by applying the Blackboard System model to a 'real-life' example. This example describes a 'groupware like', non-automated human collaboration process: the collaboratively writing of a book.

Acknowledgement

First of all, I would like to thank Dr. Luuk Groenewegen and Dr. Ida Sprinkhuizen-Kuyper for all their guidance and support, enabling me to work with constant pleasure on two topics of my interest: Blackboard Systems and SOCCA.
Furthermore, I would like to thank my parents for their ready support and encouragement during my study.
And finally, thank you Cor, for motivating me and for listening to all that abstract stuff.

Part IV: Application of the given example 60

Part I: Basic concepts

1.1 Introduction

Blackboard Systems were originally developed in the field of artificial intelligence as a method to organise problem-solving programs. The problem-solving is dynamically controlled by automated experts that communicate with each other through a global database, called 'the Blackboard' [1].
As the automated experts are in control, one of the tasks of the automated experts is to organise the problem-solving on the Blackboard. In other words, they have to enforce opportunistic evolution on the Blackboard.
This evolution can be visualised by the recording of the successive modifications on the Blackboard.

In the field of Software Process Modelling, the Blackboard System concept is of special interest as Blackboard Systems naturally provide a way to model the evolution of processes. The ability to model evolution is of great interest as it can support the structured thinking of processes.
By replacing the automated experts by humans, Blackboard Systems can also model evolving processes of human collaboration.
Not surprisingly, Blackboard Systems are already recognised in the field of CSCW (Computer Supported Cooperative Work) as a method to model and support human collaboration environments [7].

In this thesis, a Blackboard System is defined that is appropriate to model human collaboration processes.
As processes can influence and create other processes, the Blackboard System must have the capability to start up, influence and terminate other Blackboard Systems. They must be able to process concurrently and communicate with each other.
The Blackboard System model is specified by using SOCCA, a Software Process Modelling Language, that is currently still under development at the University of Leiden [4].
SOCCA is a suitable language to model this human collaboration Blackboard System, as no distinction is made between the modelling of automated and human parts. This way, the interaction between human- and automated parts or even the interaction between human parts of a system, can be modelled more explicitly than usual.
As no automated parts are included in the proposed Blackboard System, the SOCCA model will be a model of a completely non-automated process.
However, as the model provides a detailed description of the behaviour of all parts of a reproductive Blackboard System and the communication between the systems, it can also serve as the basic design of similar automated Blackboard Systems.

A SOCCA-model can become very complicated when too many details are to be modelled. For this reason, some choices have to be made. As a result, this SOCCA-model will emphasise the organisation of the problem-solving more than the problem-solving itself. However, as the experts are personified by humans, it is better to avoid too many details concerning the way problems are to be solved.

In order to visualise the evolving of and communication between human collaboration processes, the Blackboard System model is applied to a 'real life' example.
In this example, describing the process of the collaborative writing of a book, several different 'groupware-like' subprocesses can be identified, like the progress of- and decision-making during an assembly, the cooperatively working on a chapter of the book, a discussion concerning the contents of the chapter, the contracting of activities out to other groups and the evaluation of its results and finally, individual processes.

1.2 Contents

This thesis is structured as follows:
Part I introduces the main concepts that are used in this thesis like Blackboard Systems, SOCCA and the given example.
Part II presents the design of a Blackboard System that is fit to represent the evolving of human collaboration processes.
Part III presents the SOCCA model of this Blackboard System.
Part IV illustrates the evolution on the Blackboard Systems by the application of the SOCCA model to the given example. The evolution on the Blackboard Systems is represented by the means of event traces and process models.
Appendix A presents the translation of the verbal description of the example into problems and Blackboard Systems.

1.3 Blackboard Systems

The Blackboard System concept was developed by AI researchers as a method to handle organisational aspects of problem-solving programs [1] .
The idea behind the Blackboard System is first mentioned in 1962 by AI researcher Allen Newell:

> 'Metaphorically we can think of a set of workers, all looking at the same blackboard: each is able to read everything that is on it, and to judge when he has something worthwhile to add to it. This conception is just that of Selfridge's Pandemonium (Selfridge, 1995) : a set of deamons, each independently looking at the total situation and shrieking in proportion to what they see that fits their natures....[6]

Later, between 1971 and 1976, the concept was developed further during the Hearsay-II speech understanding project [9] , leading to the first Blackboard System, commonly known as the *Hearsay-II Speech-Understanding System.*

A Blackboard System consists of 3 parts:

Knowledge Sources	Independently operating software modules that have special knowledge about the problems to be solved.
The Blackboard	A global database that contains all information concerning the problems and through which the Knowledge Sources communicate with each other.
The control System	The system that determines the order in which the Knowledge Sources make changes to the Blackboard.

The advantages of this concept lie in the ability to:

- model different points of view on the problem-solving into separate modules that can behave independently of each other.
- change the knowledge involved in the problem-solving by refinement of the Knowledge Sources or by the addition of new Knowledge Sources.
- specify different problem-solving techniques into different Knowledge Sources
-dynamically control the problem-solving on the Blackboard as the Knowledge Sources are self-activating and only controlled by the Control System.

The Blackboard concept is very general and only outlines organisational principles. There is no information provided about the way working Blackboard Systems are to be developed.
Therefore, the design of a Blackboard System depends highly on the purpose of the Blackboard System.
The Blackboard System concept has proven to be a very strong and general concept that was and still is successfully applied to a large variety of problems.

Although originally designed as a method to organise problem-solving programs, the Blackboard System concept is currently also used in other ways.
By replacing the automated experts by humans, Blackboard Systems can serve very well as an organisational model of human collaboration.
As a result, the Blackboard approach is also recognised in the field of CSCW as a suitable way to dynamically control and support the processes within human collaboration environments . See also [7].

1.4 SOCCA

This section will only introduce SOCCA (Specification Of Coordinated and Cooperative Activities) briefly, as the complete description can be found in [4].
Until now, no formalism exists that is suitable to model all aspects of software processes. For this reason, SOCCA proposes a combination of 3 different formalisms to model processes:

1) The use of EER (Extended Entity Relation) diagrams to specify the data perspective. All classes and the relations between the classes that describe the static structure of the process are specified by means of EER diagrams.
   In addition to the EER diagrams, the so-called export-diagrams are used. Export-diagrams specify for every object the imported export operations of itself or other objects.

2) The use of STD's (State Transition Diagrams) to specify the first part of the behaviour perspective.
   The external and internal behaviour of the objects are defined by means of STD's. The external behaviour of an object defines the behaviour that is visible from outside, or, the allowed sequences of operation calls to the object.
   The internal behaviour of an operation, represents the functionality of the operation. It defines , the possible sequences of calls to itself or to other objects. By defining the internal behaviour of every export-operation of an object, the complete internal behaviour, or the 'hidden behaviour' of an object is defined.

3) Finally, the second part of the behaviour perspective is defined by the use of Paradigm. Paradigm (PARallelism, its Analysis, Design and Implementation by a General Method)[5] is a formalism that is based on STD's, enabling the specification of coordinated parallel processes.

By using Paradigm on top of the STD's of the external and the internal behaviour, the coordination between the internal behaviour of an object and the communication between the objects is specified.
To model this coordination, subprocesses and traps within the STD's of the internal behaviour of the objects have to be identified. A subprocess denotes temporary behaviour restrictions of the complete behaviour of an operation, a trap is a part of the subprocess that regulates the switching between the subprocesses.
The subprocesses and traps of an object are 'managed' by an STD, called the manager process. Every object has its own manager process in which possible combinations of subprocesses define the states of the object and the possible combinations of traps define the state-transitions between the objects.

1.5 The example

This thesis presents a SOCCA-model of a Blackboard System. This Backboard System is used to visualise evolution of processes on the basis of a given example, describing the collaborative writing of a book. In part IV, the actual application of the Blackboard System on the example is outlined. This section presents the example.

1.5.1 Introduction to the example

The example, the verbal description of the example is presented in the next section, originates from the second book of the PROMOTER community [3].
PROMOTER is a European project, financed by ESPRIT, in which a number of universities participate to exchange ideas on Software Process Modelling.

The example is part of chapter 7 of the book, titled 'Where will Software Process Models lead us'. It describes the actual history of the collaborative writing of chapter 7 and is used to illustrate the correspondence between Organisational Process Models and Software Process Models. In order to do so, the example is modelled in both modelling techniques.
As the choice of the example itself was not made without discussion, the objections against the example – and the refutations against these objections – are also part of the chapter.
Some of the advantages mentioned also apply to the use of the example in this thesis, for instance:

- there is much evolution
- there is a meta process
- the example is from another process world, far away from software processes
- the example describes no automated processes

The fact that the example is already analysed and modelled in the book, adds an important advantage to the use of the example in this thesis. Furthermore the example relates types of human collaboration that are so familiar to everybody that they do not need any further explanation.

1.5.2 A verbal description of the example

On 19940209 – date descriptions like this give the year, month and day in this order; so this date refers to the 9th of February, 1994 – it was being proposed in a PROMOTER meeting in Villard de Lans, France, that the PROMOTER community should start working on a second book, this second book should contain a problem-oriented presentation of the software process modelling field. After some preliminary discussions about the book structure, it was decided to prepare some proposals concerning this structure for the following day. Furthermore it was decided that the author of this

second PROMOTER book should be PROMOTER, that Jean-Claude should be the general editor, that Alfonso should be the general co-editor, and that Ali should give technical support to these editors. In addition, for every chapter to be part of the book there should be one editor, at least two authors, and two reviewers. Editors and authors were to be appointed the following day, after the decision about the (chapter) structure would have been taken.

On 19940210 there were two proposals for a possible structure. After some discussion it was decided to have a structure of the book consisting of 9 chapters. As for this example only chapter 9 – which is the present chapter 7 – is relevant, the details of the other chapters will be omitted. Chapter 9 should have the title Software Process Perspectives – an earlier version of the title actually was Related Domains. The main topic to be addressed in chapter 9 should be the question, where will software processes lead us. During a subsequent discussion, this time in groups in order to make it easier to form a team for each chapter consisting of an editor and at least two authors, Vicenzo and Luuk have formed such a small group. First they had the idea to have a preference for chapter 8, called User Interaction and Social aspects. But it was decided to prefer chapter 9, and it was moreover decided that Luuk should be the editor of that chapter, and that Jacques and Vicenzo should be the authors. As Jacques had already left Villard de Lance, it was necessary to ask him afterwards, and also to inform him about any further ideas for and possible global decisions about the chapter-to-be.
Before the discussion in small groups really started, it was also decided that each chapter team should spend some part of the evening or the night to discuss a possible set-up of their chapter, and to put the result of that discussion on 1 or 2 sheets, to be presented by each editor in the PROMOTER meeting of the following morning, in order to discuss the various set-ups. Moreover it was decided that there should be three writing and review rounds for each chapter in parallel, followed by the writing of an introduction and finishing the coherence between the parts of the book. The three rounds for writing and reviewing were also meant for enabling the various writing groups to establish a sufficient level of coherence and cross-referencing between the chapters.

During the evening discussion Vicenzo and Luuk started on the idea of having a well-chosen example as an illustrative answer to the main question of the chapter, where will software processes lead us. The very mentioning of this example triggered a whole stream of objections against it, but also the refutations of the objections. So they decided to let these objections and refutations be a substantial part of the chapter, as they certainly would be clarifying for others too. Moreover, this discussion actually led them to the formulation of a theorem, the current Theorem 7.5. As at that time they had no idea of how to prove this theorem, they did not think it probable to find a proof of it before the final version of the chapter had to be produced. So they decided, instead of proving the theorem, to give a rather thorough illustration of the theorem by presenting and discussing the example in a sufficiently instructive manner. In their opinion the collaborative writing of this chapter could very well serve as such. So formally, their theorem would have the status of a conjecture. Another point in this part of the discussion was, that by carefully considering the refutations of the objections, one might be able to find new arguments that could illustrate the theorem. In this way the part of the process where the process was being described, would lead to a better result, so it would lead to a better process than before.
On 19940211 in the full PROMOTER meeting the results of this evening discussion were reported by Luuk as chapter editor. There was an agreement on this first set-up.
Upon returning to Leiden, The Netherlands, Luuk informed Jacques in Nancy, France, about all this, and asked him whether he would like to participate. Which he liked, viewing the topic of the chapter as not an easy but an interesting challenge. His reaction too was conveyed by email, not only to Luuk, but also to Vicenzo in Pisa, Italy.

Then Luuk as the responsible editor was faced with two problems, one, how to organise the writing in more detail, especially which time period(s) should be reserved for it and who should do what, and two, how to be as illustrative as possible in representing the example, such that after the representation the theorem would look like just a straightforward abstraction step further.

Concerning the first problem, from the beginning it was the idea that Luuk should also be involved in the writing. This was actually based on an earlier writing and editing experience. Moreover, Luuk had a few sabbatical months to spend. Why not use two of these, at least partly, to get the job done. So it was arranged that in September Luuk should visit Vicenzo in Pisa, and in November Jacques in Nancy.

In trying to find an acceptable solution for the second problem, it became gradually more clear that such a maximally illustrative representation of this particular co-operative writing example should also work for other examples from a certain larger class. So the question was, what is a suitable class, and how to represent it. This actually led to the ideas expressed in Lemma 7.4 and Lemma 7.5 respectively, and thus to the idea how to prove the theorem.

Upon arriving in Pisa on 19940901 Luuk discussed this new idea of proving the theorem with Vicenzo, and they agreed upon it. The set-up of the chapter was changed accordingly, so from then on, 19940905 to be precise, the chapter was supposed to consist of more or less ten sections, the first five presenting the theory, and the last five presenting the example. It remained a somewhat open question whether the role of the example should indeed be so large as to cover the second half of the chapter. But they decided to start like this, and to judge from the result.

As Vicenzo was too heavily involved in local duties, Luuk did the writing of the first five sections while being in Pisa until 19940928, and by using Framemaker.

In the meantime Jacques was being informed about the changed set-up of the chapter. Also the general co-editor was informed. At the end of his stay in Pisa, Luuk mailed the Framemaker file both to Vicenzo and Jacques.

This finishes the relevant part of the verbal, informal description of the example.

Part II: Specification of the Blackboard System

Based on the general Blackboard System concept, a new Blackboard System is presented, that is fit to serve as a organisational model for human collaboration. This part will introduce this human collaboration Blackboard System and its features. The SOCCA-model of the Blackboard System is presented in Part III.

2.1. The basic concept of a single human collaboration Blackboard System

A *Blackboard System* (**BB-system**) contains a *Blackboard* (**BB**), a *Control Blackboard* (**CBB**), *Knowledge Sources* (**KSs**) and a *Control Knowledge Source* (**CKS**).

The purpose of a Blackboard System is to solve *problems*.
The problems to be solved are put on the Blackboard, which can be viewed as the global database of a Blackboard System.

The Knowledge Sources continuously check the BB to see if there are any unsolved problems.
In order to help solving the problems, a KS can apply his knowledge by proposing actions on an unsolved problem. The KS has to formulate the proposed actions in a *proposal* which is to be put on the Control Blackboard.

The Control Knowledge Source continuously checks the proposals on the CBB.
The CKS decides which proposals are to be executed and in what order.
When a proposal is to be executed, the CKS will activate the KS that created the proposal. The KS will then execute the proposed action on the BB.

All modifications on the BB will be registered by the CKS in the *History* on the CBB.

The KSs have special knowledge about the problems. The CKS has special knowledge about the organisation of the problem-solving activity.

2.1.1. A Blackboard System Process Model

In order to visualise the human collaboration Blackboard System, a special Blackboard System Process Model is designed. Fig 2.1 presents a Blackboard System Process Model of a single Blackboard System.
The KSs are represented by the small circles at the side of the BB. The CKS is placed at the bottom of the BB.
The unsolved problems on the BB are lined up in the top-section of the BB, the CBB is represented by a box in the bottom section of the BB.
The CBB contains the History and the proposals.

This model is used to register the state of the BB-system at a certain point of time. To show the evolving of processes, Blackboard System Process Models have to be drawn at fixed points of time.
The presented model is very simple, as it must be fit to represent a complex constellation of communicating BB-systems.
In section IV, this model is used to represent the evolving of processes as described in the given example.

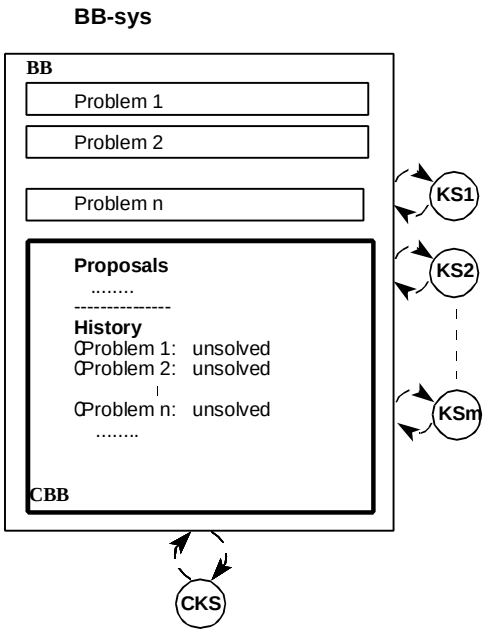


Fig. 2.1. Blackboard System Process Model of a single Blackboard System

2.2. Problems

All information on the BB is stored in the form of *problems*.
Every problem has to be defined by a *problem description* and has to be in one of three states: unsolved, solved or unsolvable. Problems in a solved state contain a 'solution' and problems in an unsolvable state have to contain a 'failure', describing the reason why the problem is unsolvable.

A KS can apply his knowledge to an unsolved problem on the BB in the form of a *modification of the BB*. The KS can choose from 3 possible modifications of the BB:

1) By putting a new subproblem on the BB

If the KS detects a 'partial' problem of the original problem, the KS can modify the BB by

- The addition of a new 'unsolved' subproblem on the BB. This way, the KS can influence the search direction of the KSs in their problem-solving activity.
  The division of an unsolved complex problem into several unsolved subproblems also simplifies the solving of the original problem

- The addition of a new 'solved' subproblem on the BB when the KS is also able to formulate its 'solution'.

- The addition of a new 'unsolvable' subproblem when the KS has detected a 'partial' problem that is unsolvable and is able to specify its 'failure'.

The creation of subproblems can be viewed as the top-down approach of problem-solving.

2) By changing the state of a problem on the BB

When the solutions of the subproblems together have solved the original problem, all useful solutions of the subproblems will be attached to the original problem and the state of the original problem will be changed from 'unsolved' to 'solved'.

If one of the vital subproblems of a problem is 'unsolvable', the state of the original 'unsolved' problem will be changed from 'unsolved' to 'unsolvable'.

The observation that the solved subproblems have 'solved' the original problem can be viewed
as the bottom-up approach of the problem-solving activity. The same applies to the observation that an 'unsolvable' subproblem has made the original problem 'unsolvable'.

3) By the deletion of a problem on the BB

In the course of problem-solving, some problems may have become superfluous or irrelevant. To avoid that KSs continue searching in these no longer relevant directions, these problems will be deleted.

If all 'initial' problems on the BB are 'solved' we can say that the BB-system is 'solved'. On the other hand, if there are no 'unsolved' initial problems left on the BB, and at least one initial problem is unsolvable, we will say that the BB-system is 'unsolvable'.

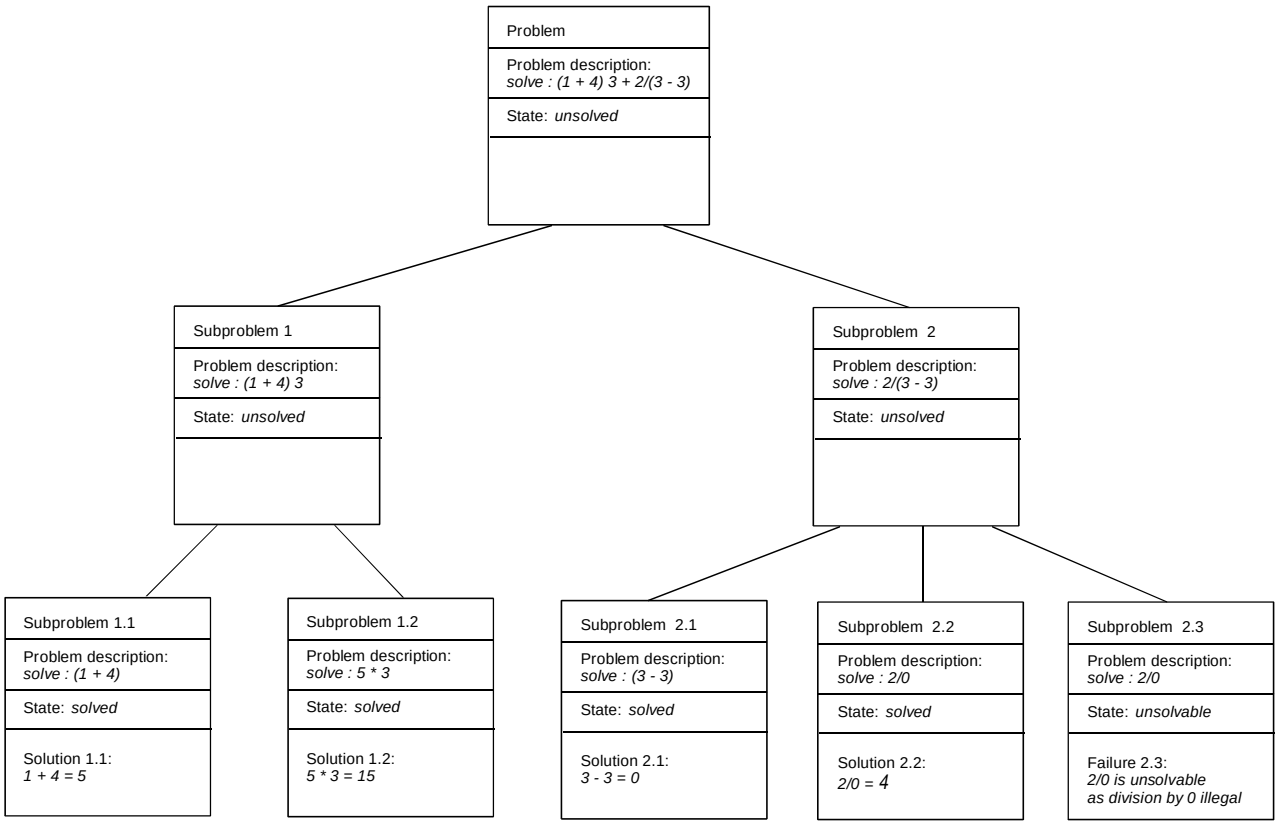


Fig. 2.2. Example: Top-down problem-solving

For example, imagine a BB-system that has to solve a simple mathematical problem.
The KSs attached to the BB-system only have the knowledge to solve 'parts' of the problem, for instance a few KSs can only add up numbers, some can only multiply numbers and some can only divide numbers. All KSs are able to divide compound mathematical statements into 'partial' statements.

First, the problem is divided into subproblems until the KSs can solve the subproblems.
Fig 2.2 illustrates this top-down problem-solving by the KSs by representing the complete tree of subproblems.

Note that subproblem 2.2 contains a false solution. This has to be recognised by the 'division' KSs and they will see to it that this subproblem is deleted.
A problem can have more than one solution. During bottom-up problem-solving, KSs may have to choose between the different solutions of a subproblem.

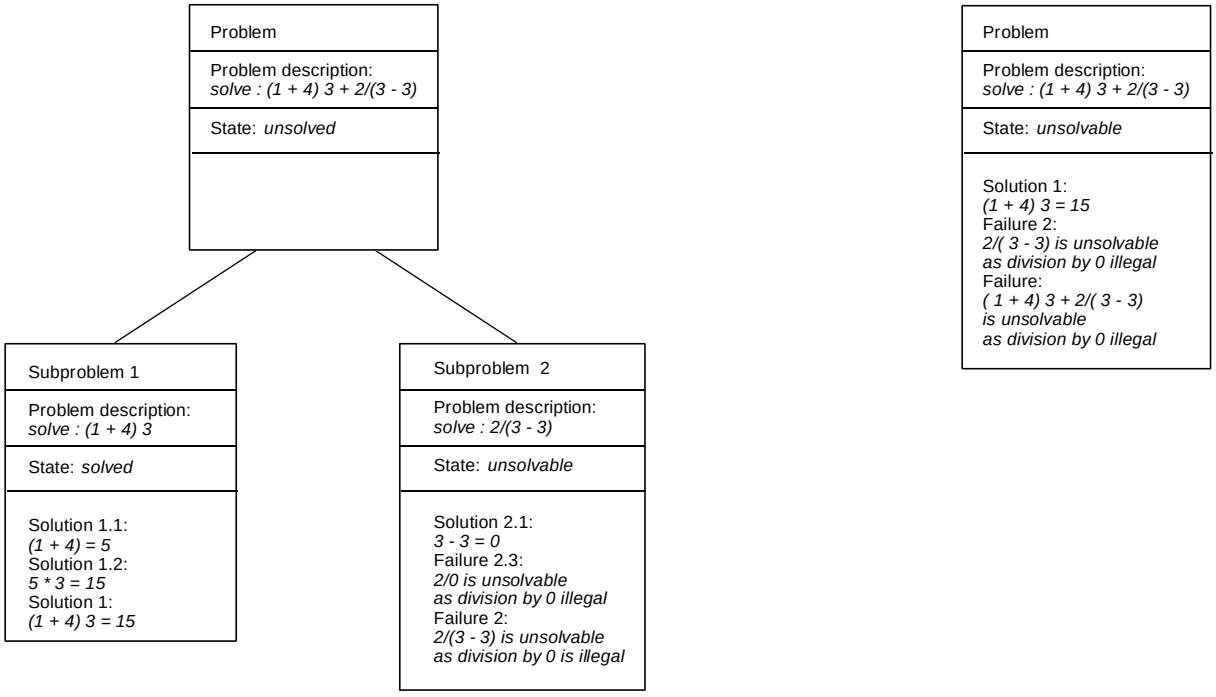


Fig. 2.3. Example: Bottom-up problem-solving.

Fig 2.3 represents two steps of the bottom-up problem-solving. The final step shows the result of the problem, containing all 'solutions' and 'failures' found during problem-solving.
Note that as soon as an essential 'unsolvable' subproblem is detected, its original problem can also be declared 'unsolvable' before the other subproblems are solved.

2.3. Child-Blackboards

A subproblem may need other KSs and CKS than those connected to the current Blackboard System. Or, they may be the same Knowledge Sources, but their roles are different.
If the creation of a subproblem affects the KSs and CKS involved, child-Blackboard Systems can be created.

For example:
A team of engineers is working on a large software project. They will be divided into smaller teams that work on specified subprojects.
Sometimes, specialists that are no member of the original team, are needed for special jobs. And some engineers may have roles in different teams.
This organisation structure can be modelled by giving each team a Blackboard System of its own.

A new child-BB-system is only generated by the creation of a new subproblem by the parent-BB-system. This subproblem will be the 'initial' problem of the child-BB-system.
Like the KSs and CKS, an 'initial' problem will be viewed as a part of the Blackboard System.

As child-BB-systems can also create their own child BB-systems, the solving of a problem may cause the creation of a tree of BB-systems. The root or very first BB-system will be called the root-BB-system. This is the only BB-system in the tree that has no parent-BB-system.
The parent of the root-BB-system will be referred to as 'outside'.

During problem-solving, the parent (or 'outside') can modify the child-BB-system (or root-BB-system) by changing its KSs or CKS or its 'initial' problems.

KSs can also modify their own BB-system. However, if a child-BB-system modifies the problem description of its own 'initial' problem, or changes its KSs or CKS, this will also affect the problem-solving of the parent-BB-system. As the parent expects an answer to the 'initial' problem by the chosen KSs and CKS, the child is only allowed to modify its own BB-system if the parent-BB-system approves with the changes to the BB-system.
This also applies to the root-BB-system: changes to the root-BB-system by its own KSs have to be authorised by 'outside'

A modification of an 'initial' problem or a change to the KSs and CKS involved is called a *modification of the BB-system*.

2.4. Proposals

Before a KS can execute any action that concerns the BB or the BB-system, the KS will have to propose this action on the CBB first.
In order to do this, the KS will create a proposal.

A KS can create a proposal for:

1) A modification of the BB (as described in section 2.2)
2) The creation of a child-BB-system
3) A modification of a child-BB-system (as described in section 2.3)
4) The termination of a child-BB-system
5) A modification of its own BB-system (as described in section 2.3)
6) The termination of its own BB-system

If the proposal is of type 1, 2, 3 or 4, and the proposal is accepted by the CKS, the KS that created the proposal, is activated by the CKS to execute the proposed actions.
The proposals of type 5 or 6 can only be activated by the parent. If the CKS of the same BB-system selects this proposal, the CKS will transfer the proposal to the CBB of the parent-BB-system.
If the BB-system is declared 'solved' or 'unsolvable', the CKS will create a *proposal for the final result* and put the proposal on the CBB of the parent-Blackboard System.
If the parent is 'outside' the results have to be related to 'outside'.

A CKS will accept any proposal of a result of a child-BB-system and put the result on its own BB as the result contains the answer of the child to an unsolved problem on the BB.

When a child-BB-system is no longer needed, KSs can propose the termination of the child. Usually, this will be done after the child has delivered its final result. But, even if the child still has unsolved problems, the child can be terminated, for instance, when the parent has 'solved' its own 'initial' problems before the child has come to a result.
Obviously, KSs of a child-BB-system can only terminate their own BB-system when this is authorised by the parent as the termination of a child also affects the problem-solving of the parent.

Note that KSs of a parent BB-system cannot propose modifications of the BB of a child-BB-system. They can only influence a child by proposing modifications of the child-BB-system.

2.5. Behaviour of the KSs

In a Blackboard System with automated Knowledge Sources, the design of the different KSs is most essential to the functioning of the BB-systems. Automated KSs may store their knowledge in rule-bases and make use of inference techniques to apply their knowledge.
Each KS involved must have his own unique knowledge and problem-solving techniques to give every KS a different view on the unsolved problems.

The proposed concept of a Blackboard System is especially designed to model human collaboration. As we can assume that humans already have their own unique knowledge and techniques, we do not have to specify this knowledge of the KSs any further.
The *role* of the KS defines what special knowledge is required.
Persons can play more than one role at the same time. For instance, it is possible for a person to be a parent, tennisplayer and programmer at the same time. The person will have to switch between these roles according to the circumstances. We will assume that persons control this switching between different roles themselves.

In a Blackboard System, a person can have more than one role. Every different role will be modelled as a separate KS.
A person can also belong to more than one Blackboard System. A KS, however, can only belong to one BB-system.
When a person is involved in different Blackboard Systems with the same role, the person will be modelled as separate KSs: one for every system.

2.6. Behaviour of the CKS

2.6.1 General behaviour of the CKS

Like the KSs, the part of the CKS is also played by a person. All properties of the KSs as described before, also concern the CKS.
In the context of the Blackboard System, more information is needed about the behaviour of a CKS.
In a Blackboard System, the KSs play the 'creative' part and the CKS the 'controlling' part of the problem-solving activity.
The complete control of the BB-system is an interaction between the KSs and the CKS.
The task of the CKS is to check whether the proposals are created by competent KSs and whether the proposed actions are legal and feasible.
The way the problem-solving takes place, depends highly on the role of the CKS.

If the CKS has the role of a chairman of an assembly, he will have to behave according to the democratic rules that belong to an assembly, in other cases, when the CKS has a more hierarchic role, he can lead the problem-solving activity in a more authoritarian way.
A CKS can also influence the amount of alternative solutions on the BB. He can lead a BB-system in a very permissive way, but he can also lead in a more restrictive way.

2.6.2 The CKS and human roles

So far, this seems to be a 'natural' way to model human roles into the KSs or CKSs. However, this modelling needs some refinement.
This refinement is needed because of the very strict distinction between the possibilities of a KS and those of a CKS.
A KS can modify the BB, but as the CKS selects the proposals, the KS can only communicate with other KSs through the CKS. On the other hand, the CKS has only indirect influence on the progress of the BB by the selection and activation of proposals.
The fact that KSs are restricted by the controlling of the CKS does only affect the problem-solving in a positive way. The CKS will see to it that the information on the BB is filtered from superfluous and incorrect information.
On the other side, the limitations of the CKS can be too restrictive to model managing human roles. For instance, a chairman does not only play a 'controlling' role, he can also play a 'leading' role in an assembly. He should be able to influence the direction of the search of the KSs in more ways than just by selecting proposals.

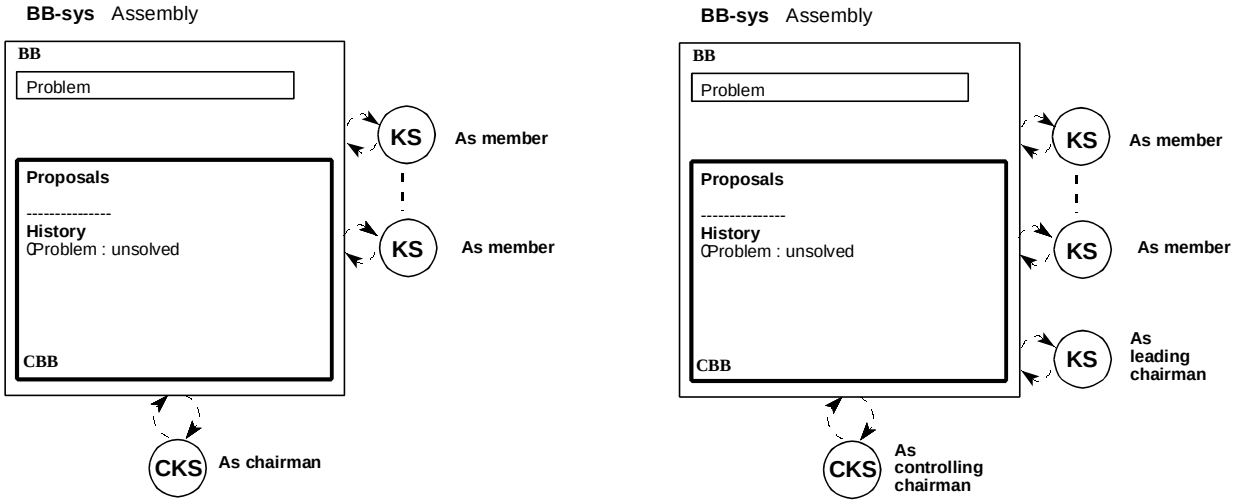


Fig. 2.4. Process Model of a Blackboard System representing an assembly. In the left Process Model, the CKS plays the role of chairman. In the right Process Model, the role of chairman is split into a role 'controlling' chairman played by the CKS and a role 'leading' chairman, played by a KS.

In this case, the role of chairman has to be split into two roles : a role of 'controlling chairman' executed by the CKS, and a role of 'leading chairman', executed by a KS (Fig. 2.2). As human roles are hardly ever definite roles, it is also possible to split up the roles of a chairman even further.

This need for distinction between separate roles within a human role is not so important when human roles are applied to KSs. A KS may play more distinct roles at the same time as long as the KS controls the switching between the different roles himself.

Note that the more restrictive a human role becomes, the more human behaviour resembles automated behaviour or the easier the human role is to be automated. As the CKS only plays a controlling role, the part of the CKS is probably the easiest part to be automated. By splitting up the human roles of the KSs likewise, the KSs too are easier to be automated.

2.6.3 Multiple CKSs in a BB-system

The proposed Blackboard System model does not exclude the use of multiple CKSs. More than one CKS can be activated by the same BB-system.
Like the KSs, the CKSs operate completely parallel, but, there is no mechanism provided to control the parallel behaviour of the CKSs. The parallel behaviour of the KSs is, in a way, controlled by the CKS.
The use of multiple CKSs probably works best when the roles of the CKSs are not overlapping.
For instance, a Blackboard System representing a project in which several persons are involved could have a CKS that controls the financial aspects of the proposals and another CKS controlling all other aspects of the proposals.
In this thesis, only BB-systems with one CKS are discussed.

2.6.4 The CKS and communication between BB-systems

The CKS also plays a major role in the communication between the BB-systems.
Of all participants in a BB-system, the CKS has the best overview concerning the state of the problem-solving activity on the BB. As the role of a CKS is to control and to monitor the problem-solving on the BB, the CKS is the most-fit Knowledge Source to control the communication between the BB-systems.
When the BB-system has solved the 'initial' problems, the CKS will put the result on the CBB of the parent-BB-system. Or, when a KS proposes an 'illegal' action, like the modification of the problem description of an 'initial' problem, the CKS will put this proposal on the CBB of the parent-BB-system.
The CKS also takes care of the input received from the child-BB-systems.

2.7 Information on the BB

Until now, there is only one possible structure to handle the information on the BB: the *problem.* Obviously, this information type cannot be sufficient in a normal working Blackboard System.

There are two objections against the addition of information types and in connection with these types, the definition of the possible modifications of these information types on the BB.
First, the proposed Blackboard System must be able to serve any possible problem-solving activity. Different problem-solving activities may require different sorts of information types to store the intermediate and final results.
Secondly, as the roles of the KSs involved in the BB-systems are played by persons, the exact knowledge of the KSs is indefinite. In connection with this property, too many details concerning the definition of information types and possible modifications of the BB by the KSs will only confine the problem-solving activity.
The KSs may invent and create their own necessary information types to store their (intermediate) results.
A possible approach for working Blackboard Systems may be the definition of a limited number of 'standard' Blackboard types as most Blackboards used in human collaboration environments will probably resemble one of these 'prototypes'.

In addition to the standard information types, other information types may be defined by the KSs involved in the BB-system.

2.8. Communication between the BB-systems

Until now, two methods of communication between BB-systems have already been discussed: the transportation of results of a child to the CBB of the parent-BB-system, the possibility to modify the child-BB-system or the request from a child to modify its own BB-system.

In addition to these methods, a BB-system is able to ask for information of all other BB-systems. A BB-system may ask for information without permission from any of its descendants: its child-, grand child-, grand grand child- systems etc. When a child-BB-system wants information of a parent, the child will receive the requested information if it is permitted to ask for information.

The proposed ways of communication between the BB-systems seems rather limited. For instance message passing and information from the 'outside world' are not explicitly modelled while they are essential to human collaboration environments.
They may not be explicitly modelled, but this information can be communicated between the BB-systems in the form of modifications of the problem description of the 'initial' problems of the Blackboard System. Information from the 'outside world' or messages can be passed to the Blackboard Systems by the modification of the root-BB-system by 'outside'. The root will pass the information to its children by the modification of these child-BB-systems and so on.
As the child-BB-systems can ask permission of their parent to modify its own BB-system, the parent is notified of the changed circumstances of the child. When the parent thinks that the proposed change also concerns its own BB-system, it can even asks its own parent to modify its own BB-system. This way, messages that concern the complete tree of BB-systems can be passed from one of the leaf-BB-systems to the root-BB-system and from the root back to all other BB-systems of the tree of existing BB-systems.

Note that information of the 'outside' world is already gathered by the CKS and KSs as they can ask for information from the 'outside' world freely and at any moment in time.

The passing of information between BB-systems by modifying a BB-system may seem a rather 'strong' way to relate information to another Blackboard System. But as only information is passed that is vital to the problem-solving of the Blackboard System, this is a correct way to deal with message passing and information from 'outside'.

Part III The SOCCA model

3.1. SOCCA

This part presents the SOCCA-model of the Blackboard System, described in Part II.

In section 3.2 and 3.3, the data-perspective of the Blackboard System is described.
Section 3.2 presents the EER-diagrams and section 3.3 presents the export-diagrams. Export-diagrams specify for every object the imported export operations of other objects or the imported export operations of itself.

In section 3.4, 3.5 and 3.6, the behaviour-perspective is described.
Section 3.4 presents the STD's of the external behaviour of the objects. The external behaviour of an object defines the allowed calling sequences of operation calls to the object.
Section 3.5 presents the STD's of the internal behaviour of the objects. The internal behaviour defines the possible sequences of calls to itself or other objects.
In section 3.6, Paradigm is applied to the STD's of the internal and external behaviour of the objects. Paradigm regulates the coordination between the internal and the external behaviour of an object and the communication between the objects.

3.2. Class diagrams

Before a model can be made, the classes involved in the SOCCA-model have to be identified. In Fig. 3.1 the classes, *BB_sys, Control System, BB, KS, CBB* and *CKS* are drawn in a class diagram.
All relations between these classes are 'part-of' relations. A 'part-of' relation is indicated by a small empty diamond at the side of the class that consists of the specified parts.
The classes *Control System, BB* and *KS* are parts of the class *BB_sys*. The classes *CBB* and *CKS* are parts of the class *Control System*.

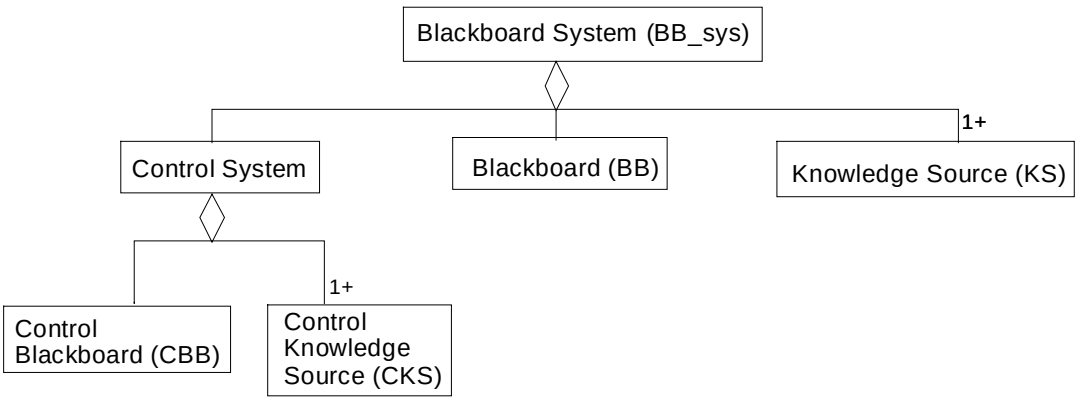


Fig. 3.1. Class diagram: classes and part-of relations

The '1+' at the side of *KS* at the relation between *BB_sys* and *KS* denotes that at least one object of class *KS* is related to one object of class *BB_sys*. All relations without the '1+' denote that exactly one object of the class is involved in the part-of relation. For instance, to an object of *BB_sys* , exactly one object of class *BB* is related.
The class diagram of Fig. 3.1 indicates that more than one CKS can be attached to a BB-system. This situation can occur, however, in this SOCCA model only BB-systems with one CKS will be discussed.

The class diagram of Fig. 3.1 shows no 'is-a' relations. In the SOCCA-model of a Blackboard System, no significant is-a relations are identified.

In Fig. 3.2, the general relationships between the classes are given.

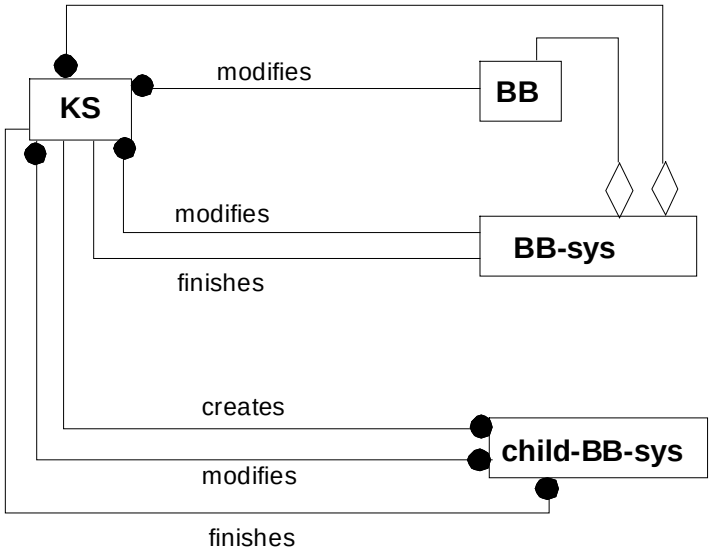


Fig. 3.2. Class diagram: classes and general relationships

A general relationship is indicated by a single line labelled with the name of the relation.
A black dot at the end of a line indicates a multiplicity of zero or more. If no dot is drawn at the end of a line, the multiplicity is exactly one.
The relations drawn in Fig. 3.2 indicate that a KS can modify only one Blackboard and that a Blackboard can be modified by zero or more KSs.
In the model of Fig. 3.2, a distinction is made between a BB-system and a child-BB-system. The classes KS and BB are part of BB-sys.
A KS can create or finish zero or more child Blackboard Systems but a child Blackboard System can only be created or finished by one KS belonging to the parent-BB-system.
A KS can modify zero or more child Blackboard Systems and a child Blackboard System can be modified by zero or more KSs belonging to the parent-BB-system.
A KS can modify his own BB-system and the BB-system can be modified by zero or more KSs belonging to the same BB-system.
Finally a KS can finish his own BB-system but a BB-system can only be finished by exactly one of the KSs belonging to the same BB-system.

Fig 3.3 shows the attributes and export operations of every class. The attributes are given in the middle section, the operations in the lower section.

The class Control System is not included in Fig. 3.3 because it plays no role in the communication between the classes.

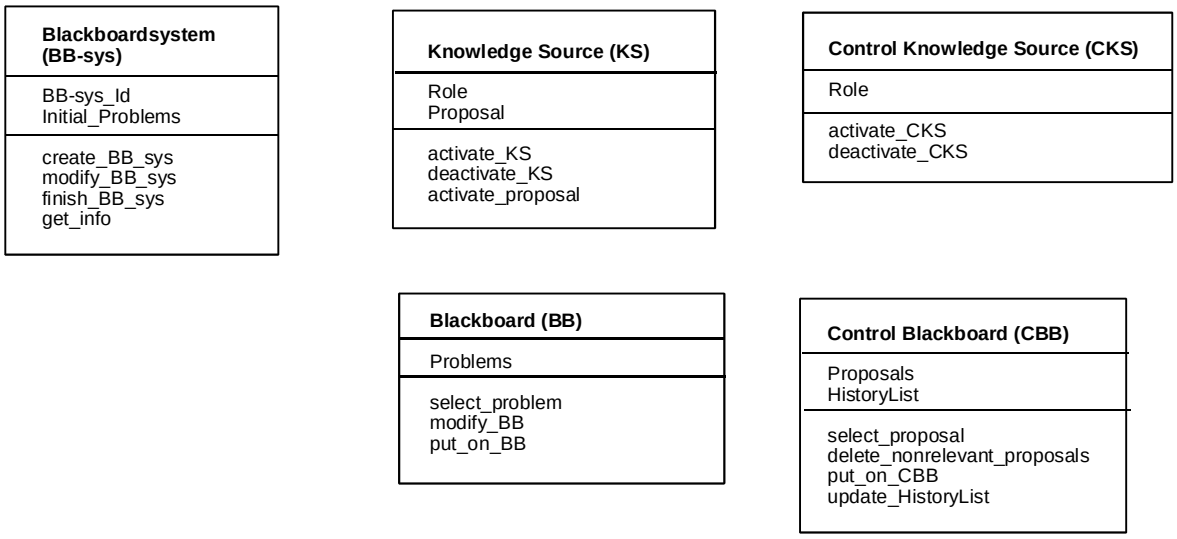


Fig. 3.3. Classdiagram: Classes with their attributes and operations

The attribute *Initial_Problems* of the class *BB_sys* contains the initial problems of the BB-system.
The export operation *BB_sys.create_BB_sys* creates and activates a new instance of the class *BB_sys*.
BB_sys.finish_BB_sys finishes, deactivates and deletes an instance of *BB-sys*.
The operation *modify_BB_sys* can be called to modify *BB_sys*
The operation *BB_sys.get_info* can be used by other BB-systems to get information about the state of the called BB-system.
The attribute *BB_sys.Permission* keeps the information concerning the BB-systems that are permitted to call the operation *BB_sys.get_info*.

The classes *KS* and *CKS* have an attribute *Role*. The role gives important information about the knowledge and behaviour of the KS or CKS.
A KS or CKS can only have one role.
The attribute *KS.proposal* contains the proposal a KS is currently creating.
The operations *KS.activate_KS* and *KS.deactivate_KS* regulate the activation and deactivation of the KS. A KS is activated at the start of a BB-system and deactivated when a BB-system is finished.
These operations can also be called in connection with a modification of the BB-system.
The operations *CKS.activate_CKS* and *CKS.deactivate_CKS* are used in the same way.
The operation *KS.activate_proposal* is called by a CKS that has chosen a proposal to be executed

The attribute *BB.problems* stores all problems on the BB. If a KS wants to select a problem, he will call the operation *BB.select_problem*. The KS can call *BB.modify_BB* to propose or execute modifications on a problem on the BB.
The operation *BB.put_on_BB* is called by a CKS to put a final result of a child-BB-system on the BB, or by a *BB-sys* to put a new or modified problem on the BB.

The attribute *CBB.HistoryList* of the class CBB stores the history of the BB-system. All actions on the BB are kept in *CBB.HistoryList* by the CKS. The CKS updates this HistoryList by calling the operation *CBB.update_HistoryList*.
The attribute *CBB.proposals* stores all proposals on the CBB.
The operation *CBB.put_on_CBB* can be called by a KS or CKS to put a proposal on the CBB of a BB-system.
The CKS can cleanup the CBB by calling the operation *BB.delete_nonrelevant_proposals*.
By calling *CBB.select_proposal*, the CKS can select a proposal on the CBB.

3.3. The Export diagram

The export diagram (Fig 3.4.) shows the uses-relations between the classes. A uses-relation specifies the export operations a class can use from another class. For instance, uses relation ‘uses 3’ indicates that *BB_sys* can call the operations *CKS.activate_CKS* and *CKS.deactivate_CKS* of class *CKS.*
The short arrows that show no particular ‘caller’, indicate that there is also another way to call the specified export operations. These operations can be called from ‘outside’. This means that there is also communication possible between a Blackboard System and ‘outside’.
This kind of communication will only take place in exceptional cases, like the creation or termination of the very first or root-Blackboard System.

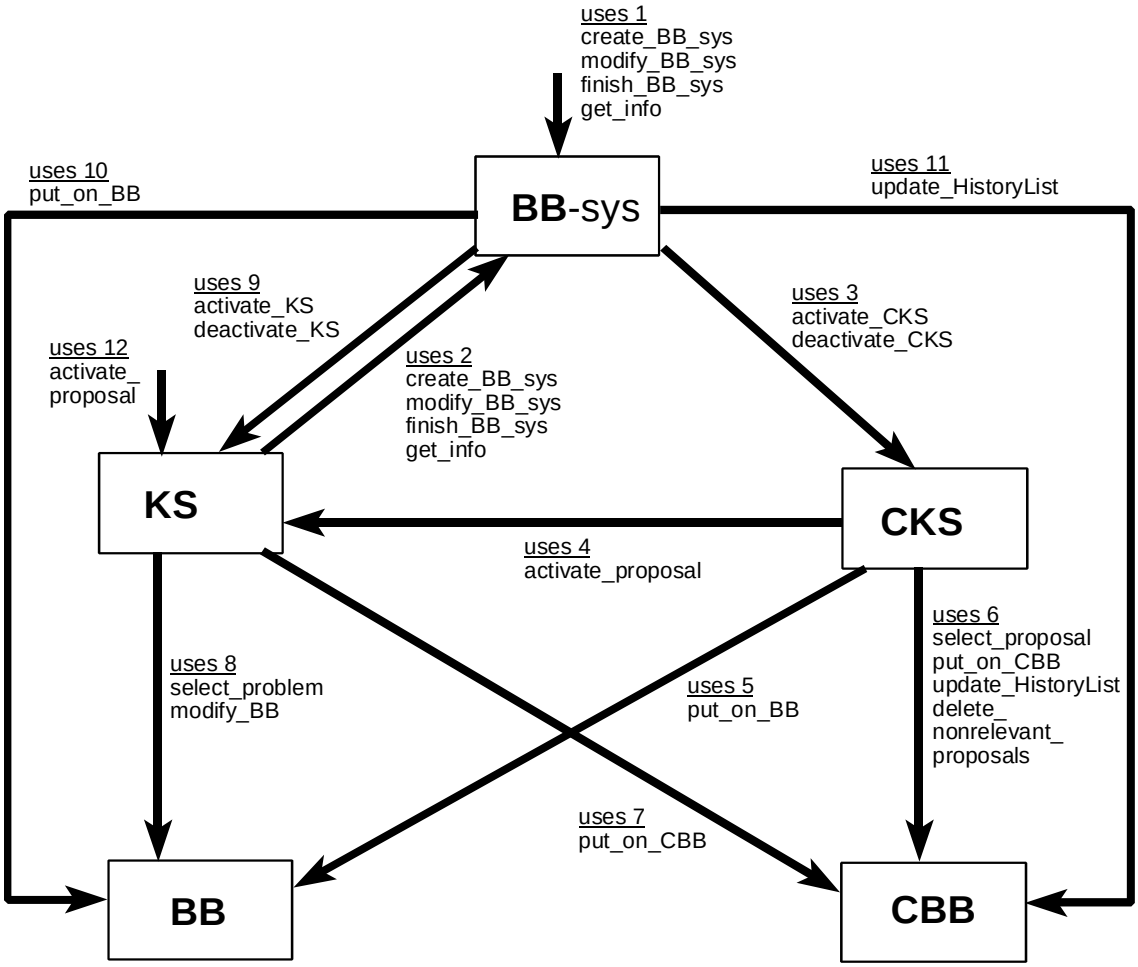


Fig. 3.4. Export diagram

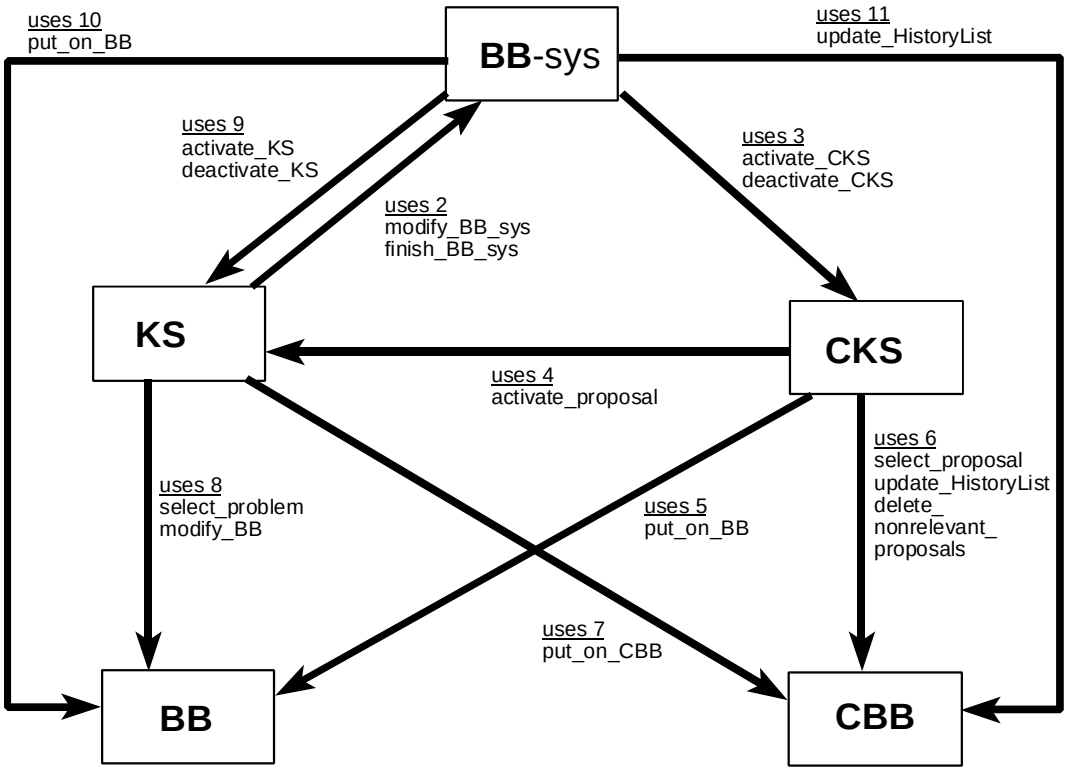


Fig. 3.5. Export diagram: Communication between the objects of one BB_sys

3.3.1. The communication between the objects of one Blackboard System.

The export diagram of Fig. 3.4, shows all possible uses-relations between the classes and between the classes and 'outside'.
Some operations, however, are only used within one Blackboard System.
Fig 3.5. presents all communication possible within one BB-system.

In Fig. 3.5, we can see that there are two operations that cannot be called within one BB-system. The export operations *BB_sys.create_BB_sys* and *BB_sys.get_info* can only be called by the parent-BB-system.

3.3.2. Communication between a parent-Blackboard System and a child-Blackboard System.

The communication between a parent-BB-system and a child-BB-system (Fig. 3.6) is a special case of communication between two Blackboard Systems.
Nearly all communication between Blackboard Systems occurs between parent- and child-systems.
The only exception on this strict parent-child communication is the operation *BB_sys.get_info*. This operation can get information from other Blackboard Systems.

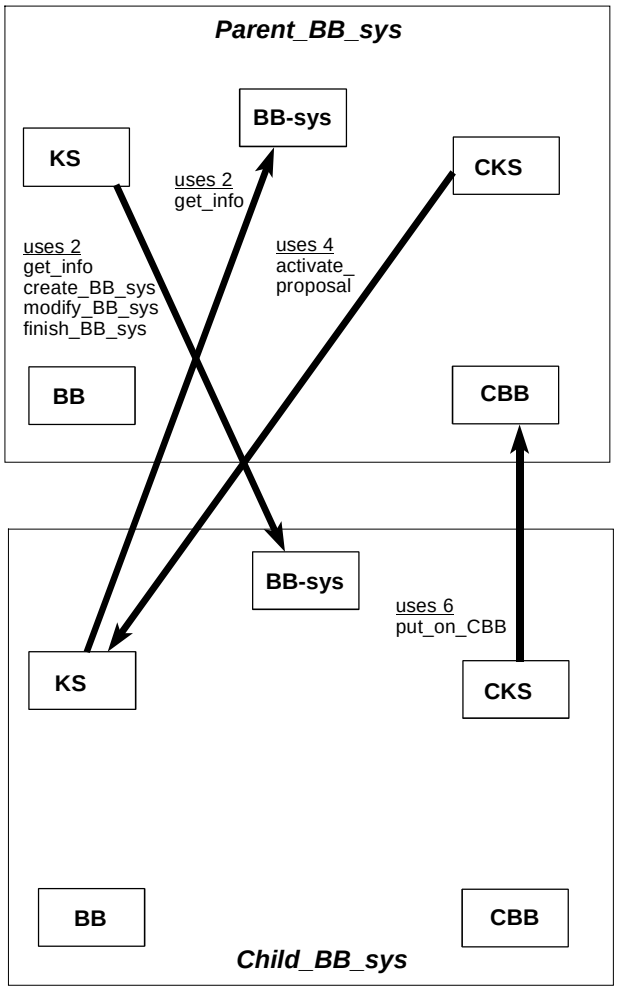


Fig. 3.6. Communication between parent- and child-Blackboard Systems

Note the similarity between the communication between a BB-system and 'outside' (Fig. 3.4) and the communication between a parent-BB-system and a child-BB-system (Fig. 3.6). 'Outside' can use the export operations *BB_sys.modify_BB_sys, BB_sys.create_BB_sys, BB_sys.finish_BB_sys* , *BB_sys.get_info* and the operation *KS.activate_proposal.* A parent-BB-system can use exactly the same operations of a child-BB-system.
The operation *CBB.put_on_CBB* is called by a child-BB-system to communicate the final result of the child-BB-system to the parent-BB-system. There is no export-operation to relate the final result of the root-BB-system to 'outside'. This will be handled by an internal operation.

3.3.3 All other communication between Blackboard Systems

In principle, it is not necessary to have this parent-child restriction for communication between BB-systems.
But, this will raise another problem : if KSs can modify the class *BB_sys* of every other BB-system, this can complicate the problem-solving activity of the systems.
To structure this complexity, some hierarchy between the BB-systems had to be defined.

3.4. STD's External behaviour

In this section, the STD's of the 'external' or visible behaviour of every class is given.
The external behaviour of a class is defined by the allowed calling sequences of its export operations and the possible states of the object.

In Fig. 3.7., the external behaviour of *BB_sys* is presented.
BB_sys has two states: 'BB_sys non existing' and 'BB_sys existing'. When *BB_sys* is in the state 'BB_sys non existing' only operation *create_BB_sys* can be called. The calling of this operation causes the state transition to the state 'BB_sys existing'. From the state 'BB_sys existing' , the operations *modify_BB_sys*, *get_info* and *finish_BB_sys* can be called. By the calling of *finish_BB_sys*, *BB_sys* will transit back to the state 'BB_sys non existing'.
The calling of the operations *modify_BB_sys* and *get_info* do not cause a state transition.

KS and *CKS* show a similar behaviour in res. Fig. 3.8 and 3.9

The classes BB and CBB only have one state called 'neutral'. From this state, all export operations can be called.

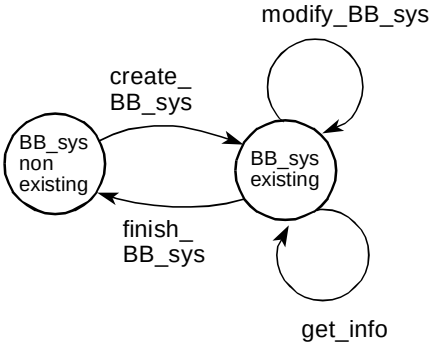


Fig. 3.7. External behaviour BB_sys

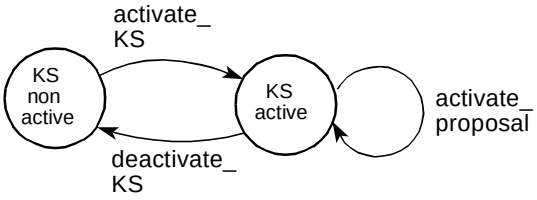


Fig. 3.8. External behaviour KS

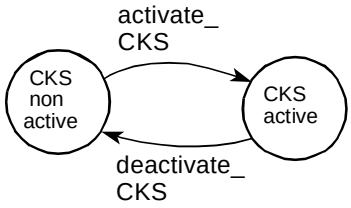


Fig. 3.9. External behaviour CKS

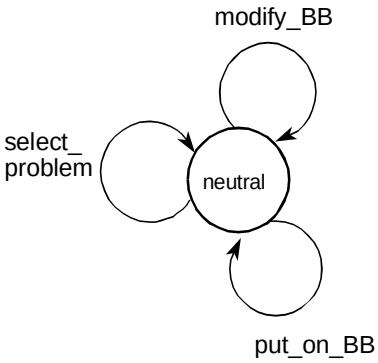


Fig. 3.10 External behaviour BB

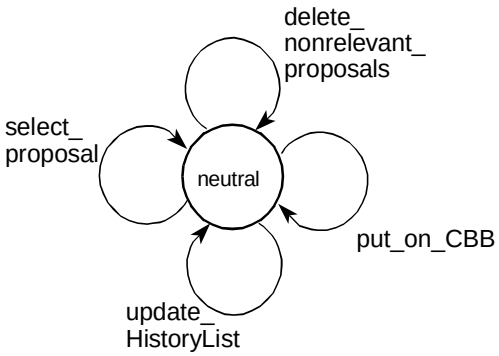


Fig. 3.11 External behaviour CBB

3.5. STD's Internal behaviour

In this section, the internal behaviour of all objects is described. The internal behaviour of an object is determined by the separate internal behaviours of its export operations.
In section 3.5.1, the STD's of the internal behaviour of the export operations of *BB_sys* are presented. The sections 3.5.2, 3.5.3, 3.5.4 and 3.5.5 present the internal behaviour of respectively *KS, CKS, BB* and *CBB.*
Within the STD's, representing the internal behaviour of the objects, two operation-types can be identified: exported operations and internal operations.
The exported operations are preceded by the word 'call'. All other operations are internal operations. An internal operation, preceded by the prefix 'act', is used to regulate communication between the external and internal behaviour of the object (section 3.6).
All other internal operations are highly internal operations within the internal behaviour of an object.

3.5.1. STD's Internal behaviour Blackboard System (BB-sys)

3.5.1.1. Operation *int-create_BB_sys*

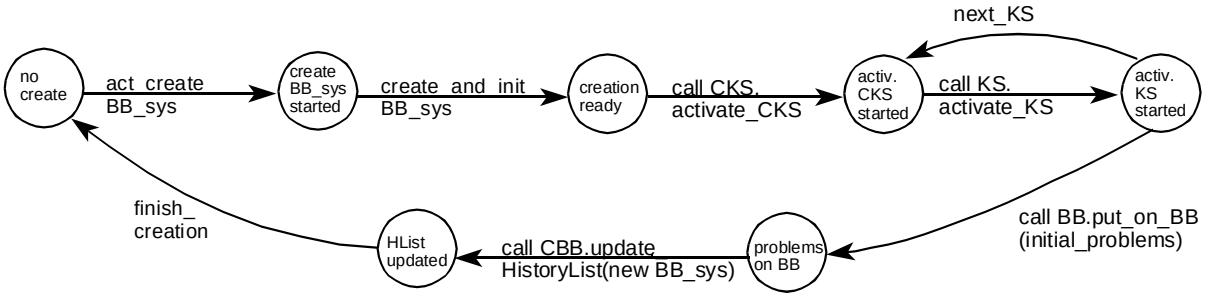


Fig. 3.12. STD internal behaviour *int-create_BB_sys*

The operation *create_BB_sys* regulates the creation and activation of a new Blackboard System.

After activation of *create_BB_sys*, the operation will proceed with the internal operation *create_and_init_BB_sys* .
create_and_init_BB_sys creates and initialises new instances of *BB_sys* and associated *BB* and *CBB.* The operation *create_and_init_BB_sys* will also initialise the new BB-system with the initial problems and KSs and CKS with their roles.

By calling *CKS.activate_CKS* and *KS.activate_KS* for every chosen CKS and KSs, the new Blackboard System is activated.

By calling *BB.put_on_BB*, the initial problems will be put on the BB.
The HistoryList of the new Blackboard System will be updated for the first time by calling *CBB.update_HistoryList*.

The operation *create_BB_sys* is called by a KS from the internal behaviour of operation *KS.activate_proposal* (3.5.2.2) of a parent-BB-system or the call for *create_BB_sys* is made from 'outside'.

3.5.1.2. Operation *int-modify_BB_sys*

The operation *modify_BB_sys* is called by a KS that is asked to activate a proposed change to a BB-system.
In principle, only a parent can create, modify or finish a BB-system. If a KS proposes a modification or the finishing of the BB-system within the same BB-system, this proposal has to be activated by the parent. The CKS has to control the correct handling of this kind of proposals.

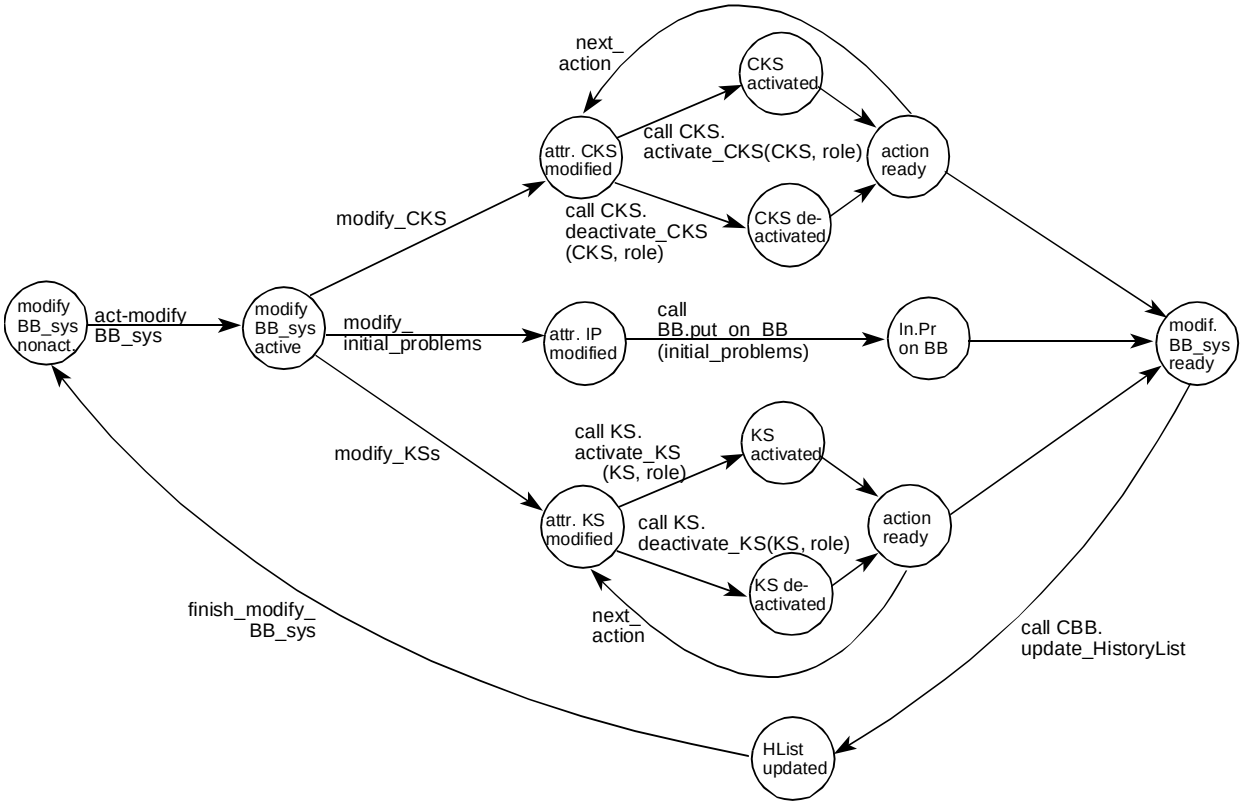


Fig. 3.13. STD internal behaviour *int-modify_BB_sys*

By calling *modify_BB_sys*, changes can be made to the KSs, CKS or the initial problems. As there is no good reason to change a BB or CBB, these parts of the BB-system cannot be changed.

After activation, the operation continues with the chosen modification, which is specified by the parameter of the operation.
If a change to the KSs is asked, the internal operation *modify_KSs* is executed. Depending on the proposed changes, a KS can be deactivated or activated more than once. For instance, if a KS has to be given an other role, the KS with the old role has to be deactivated first by calling *deactivate_KS(old_role)*. By executing *next_action*, the BB-system can continue with the activation of the KS with his new role by calling *activate_KS(new_role)*.

When the KS has finished the modifications of the BB-system, the HistoryList of the modified BB-system is updated by calling *CBB.update_HistoryList*.

The call for *modify_BB_sys* can be made by a KS from the internal behaviour of operation *KS.activate_proposal* (3.5.2.2) from within the parent-BB-system or from within the same BB-system. The call for *modify_BB_sys* can also be made from 'outside'.

3.5.1.3. Operation *int-finish_BB_sys*

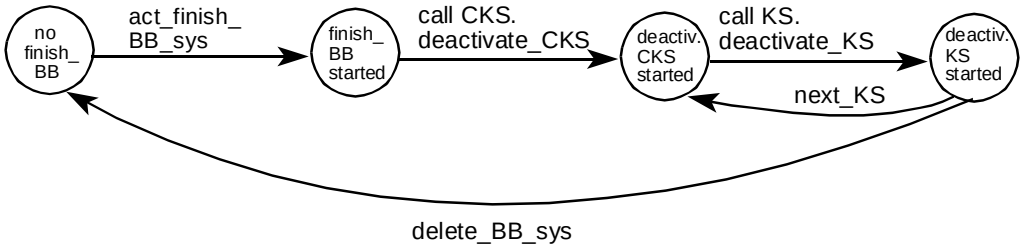


Fig. 3.14 STD internal behaviour *int-finish_BB_sys*

The operation *finish_BB_sys* (Fig. 3.14) regulates the deactivation and deletion of a Blackboard System.
The operation *finish_BB_sys* will usually be called after the child-Blackboard System has declared its initial problems solved or unsolvable and notified its parent.
If the proposal for the finishing of *BB_sys* is made by a KS of the BB-system, the proposal will have to be activated by the CKS of the parent-BB-system.

The operation *finish_BB_sys* can be called by a KS from the internal behaviour of operation *KS.activate_proposal* (3.5.2.2) from within the parent-BB-system or from within the same BB-system. The call for *finish_BB_sys* can also be made from 'outside'.
When the root-BB-system has declared its initial problem 'solved' or 'unsolvable', 'outside' is notified. 'Outside' can terminate the BB-system by calling *finish_BB_sys*. This will deactivate and delete the last or root-BB-system.

3.5.1.4. Operation *int-get_info*

This is a very simple operation. This operation will be called by a KS of another BB-system that wants information about the current state of the BB-system. The operation will only return the information when the BB-system to which the KS is connected has permission to ask for information. The permission is controlled by *BB-sys* by checking the attribute *BB_sys.Permission*.
A BB-system is permitted to receive information from any of its descendants: its child-, grand child-, grand grand child- systems etc. When a child-BB-system wants information of a parent, the child will only receive the requested information when it has the right permission.

get_info delivers the information without any further calls. So, we can omit the STD-representation of *int-get_info.*

The operation *get_info* can be called from the internal behaviour of *KS.activate_KS* or by 'outside'.

3.5.2. STD's Internal behaviour Knowledge Source (KS)

3.5.2.1. Operation *int-activate_KS*

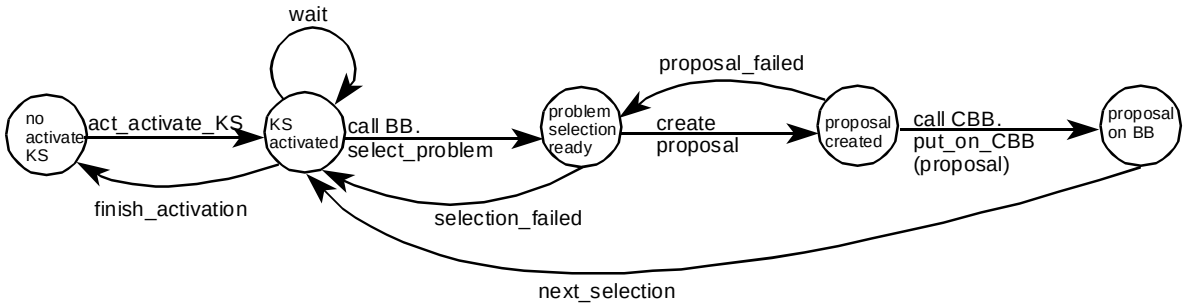


Fig. 3.14 STD internal behaviour *int-activate_KS*

There can only be ongoing activity in a Blackboard System after the KSs and CKS have been activated.
The KSs will be activated after the creation of the Blackboard System.
From that moment on, the KSs will continuously check the blackboard (BB) , by calling *BB.select_problem*, if there are problems to be solved.
If the KS cannot find a 'fit' problem, the KS will execute the internal operation *selection_failed* and check the BB later.

If the KS has found a problem, the operation *BB_select_problem* will make a copy of the chosen problem. By making a copy of the problem, the problem itself will remain available and unchanged on the BB for other KSs during proposal-creation and proposal-selection.
The KS will create a proposal for the copied problem by executing the internal operation *create_proposal*.
In the proposal, actions on the copied problem can be defined.
If the proposal fails, the internal operation *proposal_failed* is executed.

If the proposal is created successfully, the KS will put the proposal on the CBB of his own BB-system by calling *CBB.put_on_CBB.*

Although not represented in the STD, the call for the operation *BB_sys.get_info* is also made from *int-KS.activate_KS. BB_sys.get_info* can be called from all states of *int-activate_KS* except the state 'no activate KS'. As the calling of *BB_sys.get_info* does not cause any state-transition, the representation of this calling is left out.

To regulate the continuous checking of the BB by KSs, the KS can make use of the operation *wait*.
Usually, the STD of an operation only specifies all possible sequences of events that determine the behaviour of the operation. There is no information given about the time an operation will remain in a specific state.
In this special case, we want to be more explicit about the time a KS will remain in the state 'KS activated' before he continues with calling the operation *BB.select_problem.*
In a BB-system, many KSs can be involved. They all continuously check the BB by calling *BB.select_problem*. As this continuously checking of the BB may affect the ongoing activity on the BB, we may want the KS to 'wait' before he checks the BB again, especially when a KS has just executed the internal operation *selection_failed*.

After the execution of the wait-function, the KS can decide to 'wait' even longer.
A KS will remain active until termination of the BB-sys or until the KS is deactivated in connection with a modification of the BB-system.
The call for *activate_KS* will be made from the internal behaviour of *BB_sys.create_BB_sys* (3.5.1.1) or *BB_sys.modify_BB_sys* (3.5.1.2.)

3.5.2.2. Operation *int-activate_proposal*

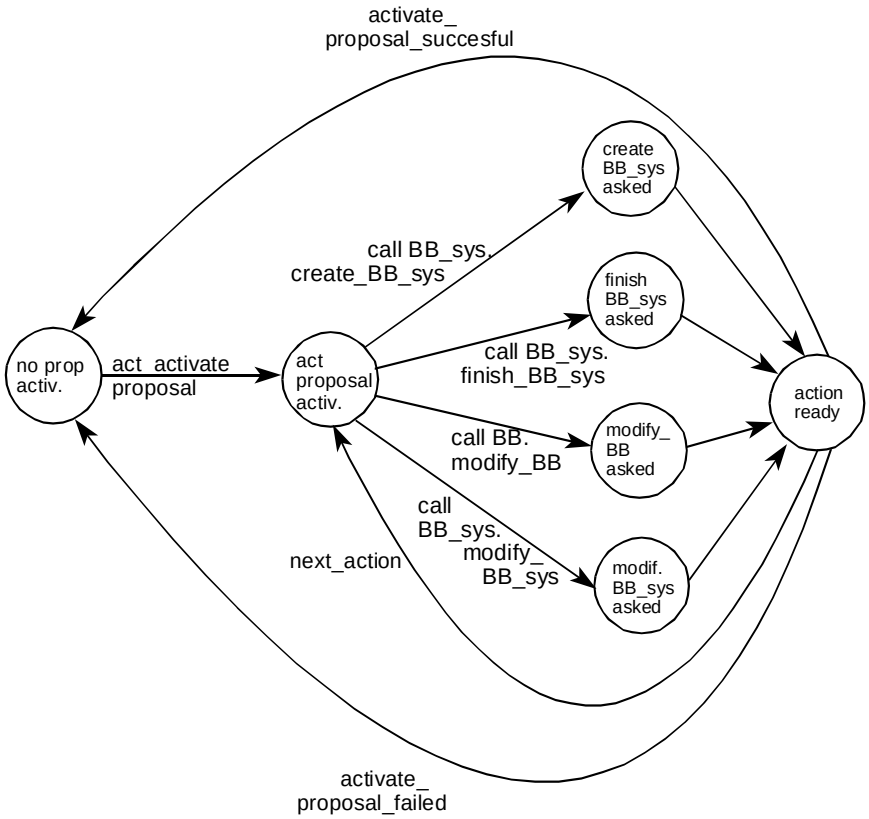


Fig. 3.15. STD internal behaviour *int-activate_proposal*

The operation *activate_proposal* is called by the CKS when the CKS has selected a proposal. By calling this operation, the KS that created the chosen proposal will activate the proposed actions on the original problem. The proposed actions are specified by the use of a parameter of the operation.

A proposal can contain one or more of the following actions:

(1) a change of the state of the problem, for instance, the change of state 'unsolved' to 'solved'.
(2) the deletion of the problem on the BB.
(3) the addition of a subproblem of the problem on the BB
(4) the creation of a new child-BB-system to solve a subproblem of the problem
(5) a modification of a BB-system
(6) the termination of a child-BB-system

The actions (1), (2) and (3) are executed by calling *BB.modify_BB*,
the action (4) is executed by calling *BB_sys.create_BB_sys*,
the action (5) is executed by calling *BB_sys.modify_BB_sys*

and the action (6) is executed by calling *BB_sys.finish_BB_sys*.
Before a proposal is activated, the CKS will have to check the HistoryList to make sure that the proposed actions of a chosen proposal do not conflict with proposals that are already activated.
As it is very unlikely that a proposed action fails, no special precautions are taken to handle failed actions.
If the activation of a proposed action fails, the failure will be registered by the CKS in the HistoryList on the CBB.

The call for *activate_proposal* will be made from the internal behaviour of *CKS.activate_CKS* (3.5.3.1) belonging to the parent-BB-system or the same BB-system.

KS.activate_proposal can also be called from 'outside' in case the BB-system concerned is the root-BB-system.

3.5.2.3. Operation *int-deactivate_KS*

This is a very simple operation, called by *BB_sys* .
This operation is called when the BB-system is finished or when the KS is deactivated as a result of a modification of the BB-system.

As no calls for other export operations are made from the internal behaviour of *KS.deactivate_KS,* we will omit a STD for the internal behaviour of *int-deactivate_KS.*

The call for *deactivate_KS* will be made from the internal behaviour of *BB_sys.finish_BB_sys* (3.5.1.2.) or *BB_sys.modify_BB_sys.*

3.5.3 STD's Internal behaviour Control Knowledge Source (CKS)

3.5.3.1. Operation *int-activate_CKS*

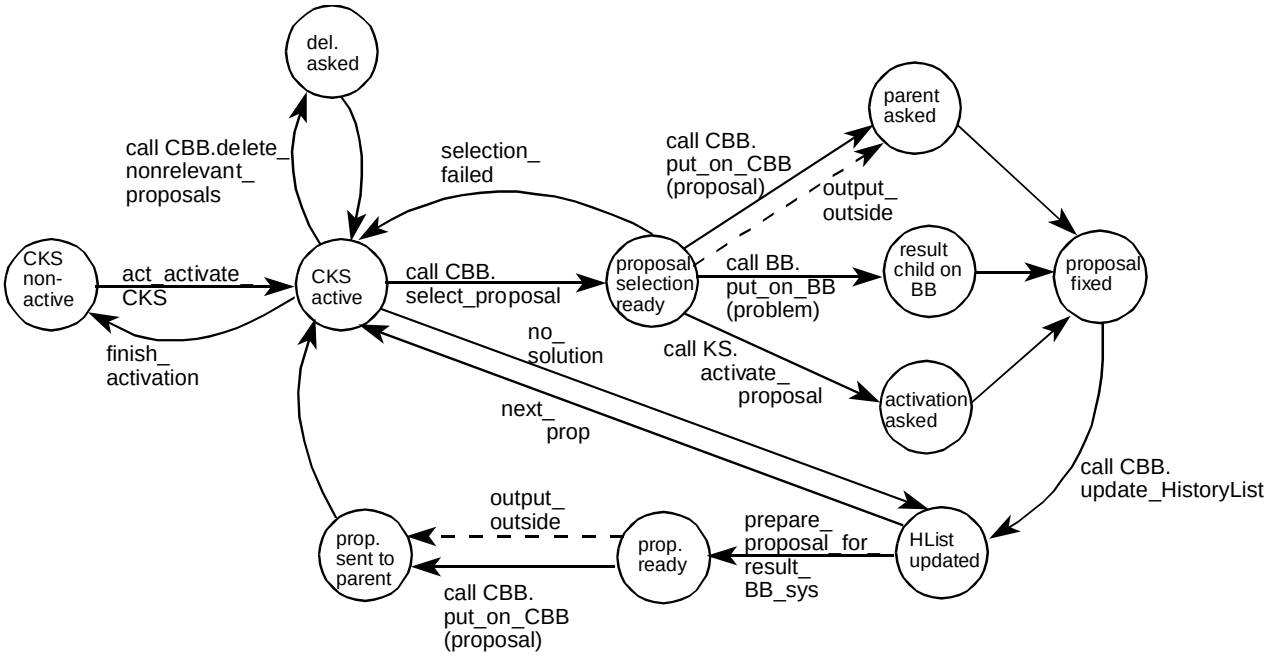


Fig. 3.17. STD internal behaviour *int-activate_CKS*

Like the KSs, the CKS is activated after the creation of the BB-system.
After activation, the CKS will try to select a proposal on the CBB by calling operation *CBB.select_proposal*. If the selection fails, the CKS will go back to the previous state by executing the internal operation *selection_failed* and try again later.
If the selection is successful, the CKS will have to decide what to do next:

(1) : If the selected proposal is a proposal for the result (solution or failure) of a child-BB-system, created by the CKS of a child-BB-system, the CKS will call the operation *BB.put_on_BB* to put the received result on the BB.

(2) If the selected proposal is a proposal for the modification or the termination of the current BB-system, this proposal can only be activated by the parent of the BB-system.
In this case, the CKS will call *CBB.put_on_CBB* of the parent-BB-system to put the proposal on the CBB of the parent-BB-system.
If there is no parent-BB-system, the BB-system in question is the root-BB-system the result has to be related to 'outside'. In this case, instead of the operation *CBB.put_on_CBB* , the internal operation *output_outside* will be called.

(3) In all other cases, the proposal will be activated by calling *KS.activate_proposal* of the KS that created the proposal.

The CKS will continue with the call for *CBB.update_HistoryList*, in which all actions are kept. By doing this, the CKS can keep track with the state of his own Blackboard System. For instance, if all initial problems are solved, the CKS will know that his own BB-system is 'solved'.

When the CKS receives no 'fit' proposals, the CKS can decide, after a certain period of time, that the initial problem(s) is (are) unsolvable. The CKS will then execute the internal operation *no_solution*. If the BB-system has arrived in a solved or unsolvable state, the CKS will prepare a proposal for the result of the BB. This proposal, will be put on the CBB of the parent-BB-system.
Again, if there is no parent-BB-system, the result has to be related to 'outside'. In this case, instead of the operation *CBB.put_on_CBB* , the internal operation *output_outside* will be called.

A CKS will remain active until termination of the BB-system or until the CKS is deactivated in connection with a modification of the BB-system.

The CKS can clean up the CBB by deleting nonrelevant proposals from the CBB by calling the operation *CBB.delete_nonrelevant_proposals*.

The call for *activate_CKS* will be made from the internal behaviour of *BB_sys.create_BB_sys* (3.5.1.1) or *BB_sys.modify_BB_sys* (3.5.1.2).

3.5.3.2. Operation *int-deactivate_CKS*

deactivate_CKS is a very simple operation, called by *BB_sys* .
This operation is called when the BB-system is finished or when the CKS is deactivated as a result of a modification of the BB-system.

As no calls for other export operations are made from the internal behaviour of *CKS.deactivate_CKS*, we will omit the STD of the internal behaviour of *int-deactivate_CKS*.

The call for *deactivate_CKS* will be made from the internal behaviour of *BB_sys.finish_BB_sys* (3.5.1.2.) or *BB_sys.modify_BB_sys* (3.5.1.2).

3.5.4. STD's Internal behaviour Blackboard (BB)

3.5.4.1. Operation *int-select_problem*

The operation *select_problem* is called by a KS in order to select a problem.
In fact, a KS has to check the BB before the KS can select a problem. This checking of the BB is not explicitly modelled, in order to simplify the model.

As no calls for other export operations are made from the internal behaviour of *BB.select_problem*, we will omit a STD for the internal behaviour of *BB.select_problem* .

The call for *BB.select_problem* is made from within the internal behaviour of *KS.activate_KS* (3.5.2.1) that belongs to the same BB.system.

3.5.4.2. Operation *int-modify_BB*

The operation *modify_BB* is called by a KS that is asked to execute a proposed action on the BB.
By calling the operation *modify_BB*, one of the following actions can be executed on the BB:

(1) a change of the state of a problem, for instance, the change of state 'unsolved' to 'solved'.
(2) the deletion of a problem on the BB.
(3) the addition of a subproblem on the BB

As no calls for other export operations are made from the internal behaviour of *BB.modify_BB*, we will omit a STD for the internal behaviour of *BB.modify_BB.*

The call for *BB.modify_BB* is made from within the internal behaviour of *KS.activate_proposal* (3.5.2.2), that belongs to the same BB-system.

3.5.4.3. Operation *int-put_on_BB*

This is a simple operation. It puts a problem on the BB.
As no calls are made from the internal behaviour of *int-put_on_BB*, we will leave out the STD of this operation.

The call for this operation will be made from within the internal behaviour of *CKS.activate_CKS, BB_sys.create_BB_sys* or *BB_sys.modify_BB_sys* that belongs to the same BB-system.

3.5.5. STD's Internal behaviour Control Blackboard (CBB)

All operations of class *CBB* are very simple operations. As no calls for other export operations are made from the internal behaviour of the operations of *CBB*, we will omit the STD's of the internal behaviour of the operations of *CBB*.

3.5.5.1. Operation *int-select_proposal*

The operation *select_proposal* is made by a CKS that wants to select a proposal.
Like the operation *BB.select_problem*, the checking of the *CBB* by the *CKS* before the selection is made, is not included in this model.

The call for *select_proposal* is made from within the internal behaviour of operation *CKS.activate_CKS* (3.5.3.1.) that belongs to the same BB-system.

3.5.5.2. Operation *int-put_on_CBB*

The call for *put_on_CBB* is made from within the internal behaviour of operation *KS.activate_KS* (3.5.2.1.) of the same BB-system or from within the internal operation of *CKS.activate_CKS* (3.5.3.1.) of the parent-BB-system or the same BB-system.

3.5.5.3. Operation *int-update_HistoryList*

The call for *update_HistoryList* is made from within the internal behaviour of operation *BB_sys.create_BB_sys* (3.5.1.1.), *BB_sys.modify_BB_sys* (3.5.1.2.) or from within the operation *CKS.activate_CKS* (3.5.3.1.) Both calls will be made from within the same *BB_sys*.

3.5.5.4. Operation *int-delete_nonrelevant_problems*

The call for *delete_nonrelevant_proposals* is made from within the internal behaviour of operation *CKS.activate_CKS* (3.5.3.1.) of the same *BB_sys*.

3.6. Subprocesses and traps

The STD's of the external and internal behaviour only describe the sequential behaviour of the objects. To regulate the interaction between the internal and external behaviours of the objects and the communication between the objects, Paradigm is used.
The STD of the external behaviour of an object serves as the manager process of all internal behaviours of this object as well as manager process of all internal behaviours from other objects that call some operation provided by this manager in its external behaviour.
The internal behaviours are called the 'employees' of the manager process.
The manager process prescribes all permitted state transitions of the employees, however, as the states of these combined behaviours determine the state of the manager process, we can also say that the 'manager' is managed by its 'employees'.
To coordinate the parallel behaviour of the employees, subprocesses and traps are used.

3.6.1. Subprocesses with respect to Blackboard System (*BB_sys*)

In Fig. 3.23., the manager process of *BB_sys* is presented.
In this subsection, we first present the subprocesses and traps with respect to the activation of the export operations of *BB_sys*. Next, the subprocesses and traps with respect to the calling of the export operations of *BB_sys* are presented.

Subprocesses and traps in connection with the activation of the operations of *BB_sys*:

The subprocesses S 1 and S 2, and traps T 1 and T 2 of the operation *BB_sys.create_BB_sys* with respect to *BB_sys*, are presented in Fig. 3.18.
If the operation *create_BB_sys* is called and its internal behaviour is in subprocess S 1 and also in trap T 1, *BB_sys* can transit from subprocess S 1 to S 2.
BB_sys will then go through all states of *int-BB_sys.create_BB_sys* until T 2 is entered.
When T 2 is entered, *BB_sys* can now transit back to subprocess S 1 where the operation can be finished.
Note that T 2 coincides with the last state before the operation *BB_sys.create_BB_sys* is finished. In this state, the new *BB_sys* is already created and activated.
If T 2 had been chosen as large as possible, conflicts between the behaviours of *BB_sys* could arise. For instance: if *BB_sys.finish_BB_sys* is called immediately after the calling of *BB_sys.create_BB_sys*, *BB_sys* could call *KS.deactivate_KS* before *KS.activate_KS* is called.

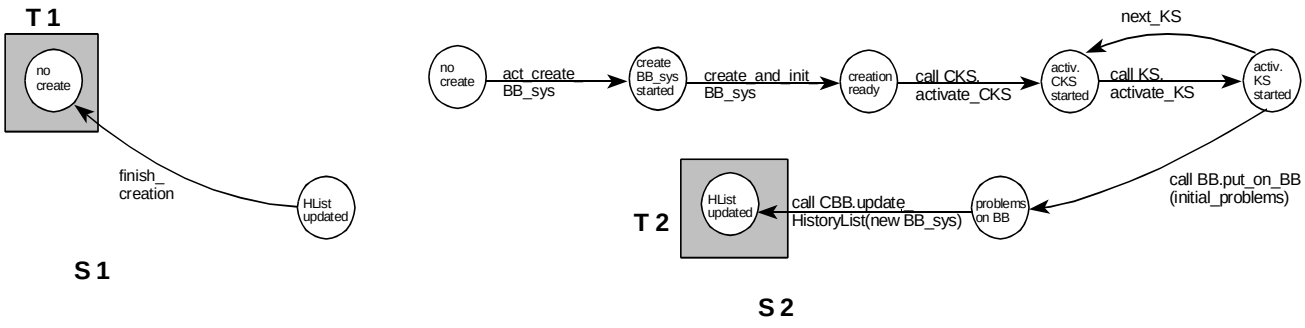


Fig. 3.18. S 1 and S 2 : subprocesses of *int-BB_sys.create_BB_sys* with respect to *BB_sys*

Fig 3.19. and 3.20. represent the subprocesses and traps of respectively the internal behaviours of the operations *BB_sys.modify_BB_sys* and *BB_sys.finish_BB_sys*.

The traps are chosen the same way as the traps of the operation *BB_sys.create_BB_sys*: T 4 coincides with the last state before termination of the internal behaviour of the operation.
This way the activation and deactivation of the KSs and CKS are co-ordinated in a correct way.
The trap T 6 also makes sure that the manager process of *BB_sys* cannot arrive in state 'BB_sys not existing' before *BB_sys* is actually deactivated and deleted.

Fig 3.21. represents the subprocesses and traps of the operation *int-BB_sys.get_info*. This time, trap T 8 is chosen as large as possible. This way the manager process of *BB_sys* can go back to the state 'BB_sys existing' as soon as the internal behaviour of *BB_sys.get_info* is started.
Other operations can be started when the manager process has arrived back in the state 'BB_sys. existing'.

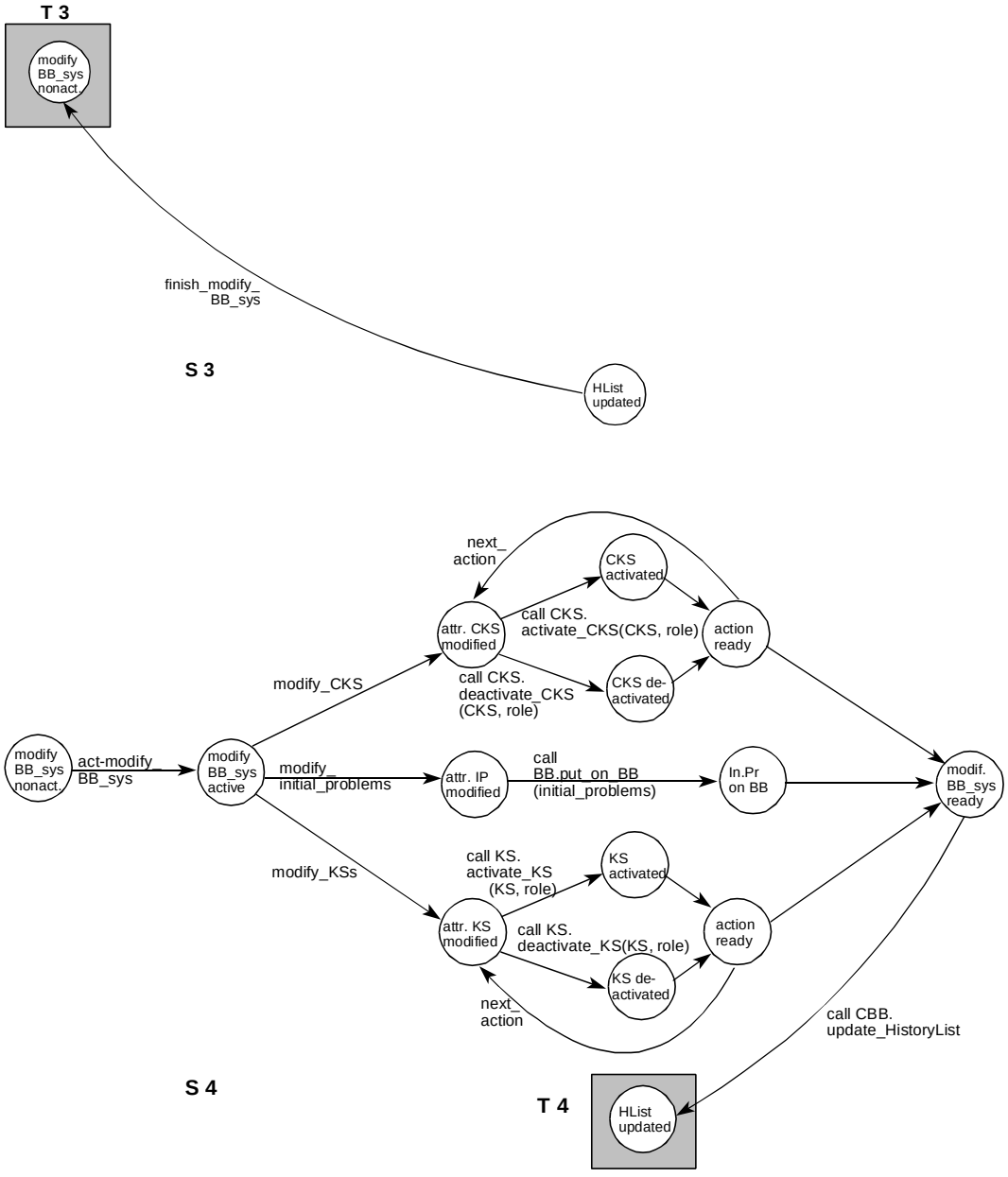


Fig. 3.19. S 3 and S 4: subprocesses of *int-BB_sys.modify_BB_sys* with respect to *BB_sys*

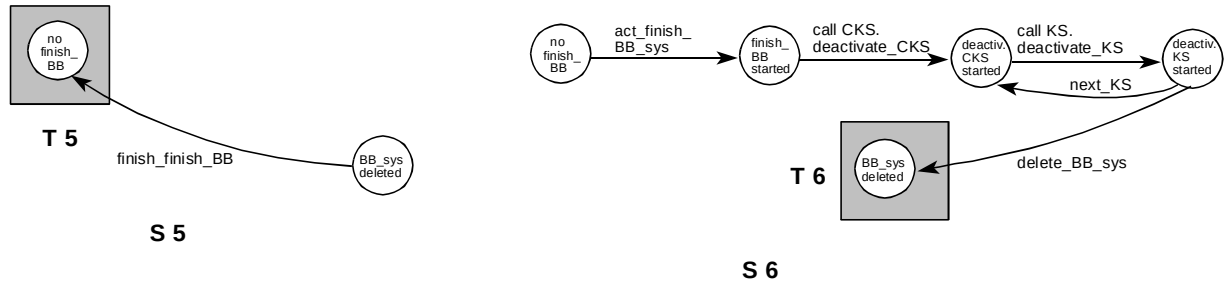


Fig. 3.20. S 5 and S 6 : subprocesses of *int-BB_sys.finish_BB_sys* with respect to *BB_sys*.

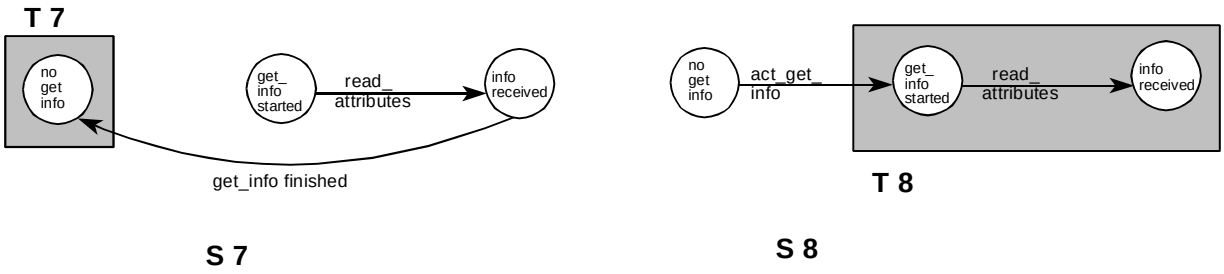


Fig. 3.21. S 7 and S 8 : subprocesses of *int-BB_sys.get_info* with respect to *BB_sys*.

The subprocesses and traps with respect to the activation and finishing of the internal behaviours of the export operations are in this SOCCA-model always chosen in one of the two presented ways:
The trap of the subprocess representing the activated behaviour is chosen:

(A1) as large as possible when the internal behaviour of the operation does not interfere in an illegal way with other operations or

(A2) the last state before the finishing of the internal behaviour if the operation has to be finished before other operations can be called.

And the trap of the subprocess representing the terminating of the behaviour contains the nonactive state of the behaviour.

Subprocesses and traps in connection with the calling of the operations of *BB_sys*:

The subprocesses and traps of the internal behaviour of 'caller' operation *KS.activate_proposal* of KS within a parent-BB-system are given in Fig. 3.22.
From the internal behaviour of *KS.activate_proposal* from within the parent-BB-system, the operations *BB_sys.create_BB_sys*, *BB_sys.modify_BB_sys* and *BB_sys.finish_BB_sys* can be called. *BB_sys.modify_BB_sys* and *BB_sys.finish_BB_sys* can also be called from within the same BB-system.

In Fig. 3.22. the subprocesses and traps concerning the calls from within the parent-BB-system are presented.

BB_sys will arrive in trap T 9 when the call for *BB_sys.create_BB_sys* is made by the parent. When *BB_sys.create_BB_sys* is in T 1, the transit can be made from S 9 to S 11.

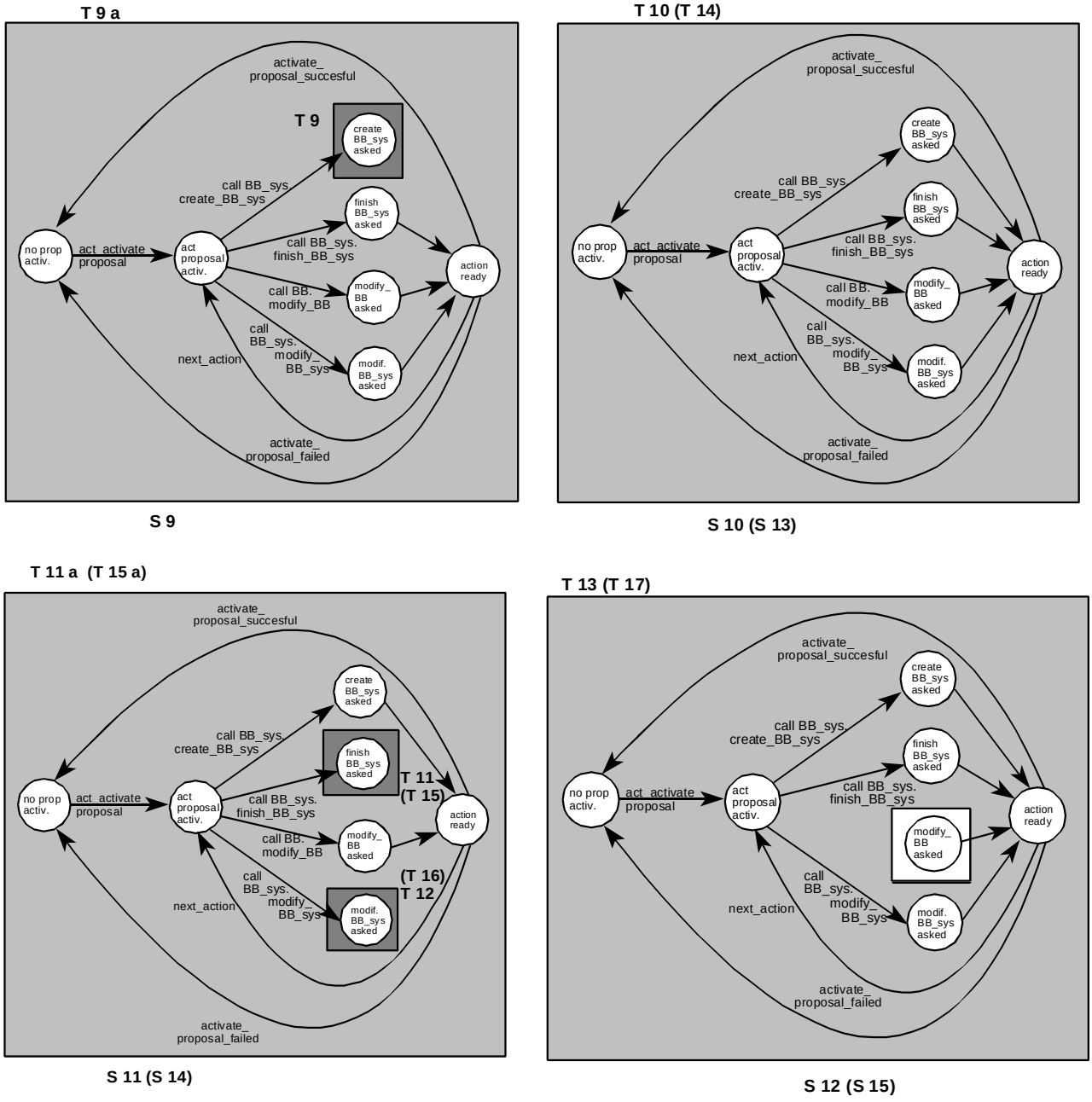


Fig. 3.22. S 9, S 10, S11 and S 12: subprocesses of *int-KS.activate_proposal* within the parent BB-system with respect to *BB_sys.*
The subprocesses and traps of *int-KS.activate_proposal* within the same BB-system with respect to *BB_sys* are very similar. S 13 , S 14, and S 15 with traps T 14, T 15, T 15a, T 16 and T 17 are exactly the same as resp. S 10, S 11, and S 12 with traps T 10, T 11, T 11a, T 12, and T 13. As a BB-system cannot create itself, the subprocess that corresponds with S 9 is S 13.

When *BB_sys* is a root-BB-system, *BB_sys.create_BB_sys* is called from 'outside'. In this case, *int-KS.activate_proposal* has to transit from S 9 to S 10 as no calls can be made for *BB_sys* from a parent-BB-system. As a transition from one subprocess to another can only be made when the subprocess has entered a trap, an additional trap is needed. This trap, T 9a, is used to force the transition from S 9 to
S 10 before subprocess S 9 has reached T 9. Note, that subprocess S 9 contains a nested trap.

In S 11, the call for *BB_sys.create_BB_sys* does not need a trap, as *BB_sys* is already created. Only calls for the creation of other instances of *BB_sys* can be made.
In S 11, calls can be made for *BB_sys.modify_BB_sys* and *BB_sys.finish_BB_sys*.
When *BB_sys.finish_BB_sys* is called, *int-KS.activate_proposal* will arrive in T 11. When *int-BB_sys.finish_BB_sys* is in T 5, the transition can be made from S 11 to S 10, where only other instances of *BB_sys* can be called. When BB_sys is in T 10 and T 6, the transition can be made back to S 9 and S 5, where *BB_sys* is no longer existing .

As the operations *BB_sys.modify_BB_sys* and *BB_sys.finish_BB_sys* can also be called by a KS from within the same BB-system, 3 subprocesses are added that resemble the subprocesses presented in Fig. 3.22. very much.
S 13 with trap T 14 will be exactly the same as S 10 and T 10 : no calls can be made in connection with *BB_sys* if *BB_sys* is not existing.
S 14 with traps T 15, T 15a and T 16 will be exactly the same as S 11 and T 11, T 11a and T 12: when *BB_sys* is existing, calls can be made for the operations *BB_sys.modify_BB_sys* and *BB_sys.finish_BB_sys*.
The traps T 11a and T 15a are needed to regulate the calling for *BB_sys.finish_BB_sys* by two different BB-systems. When *BB_sys.finish_BB_sys* is called by the parent, *KS.activate_proposal* of a KS of the child will also have to transit from S 14 to S 13 before reaching a trap as the child cannot make any calls for a *BB_sys* that is no longer existing. In this case T 15a will be used to transit from S 14 to S 13.
When *BB_sys.finish_BB_sys* is called by a child, T 11a is used to force S 11 to transit to S 10.
S 15 with trap T 17 is exactly the same as S 12 and T 13.
As a BB-system cannot create itself, the subprocess corresponding with S 9 will be S 13.

The subprocesses S9-S12 all concern the behaviour of one *KS* of the parent-BB-system and S13-S15 concern the behaviour of one *KS* of the same BB-system.
In fact, several KSs can be involved in the parent BB-system. The BB-system itself can also have several KSs attached to it. As all KSs involved behave in parallel, all KSs should have their own subprocesses in the manager process of *BB-sys*.
To simplify the manager process of *BB_sys*, the subprocesses that are to be multiplied in case of more than one KS, are indicated by the symbol '*' in the manager process.

The operation *BB_sys.get_info* is called from within the internal behaviour of *KS.activate_KS*. As long as *KS* is activated, the *KS* is free to call this operation at any point in time. For this reason, it is not necessary to include any subprocesses and traps with respect to the calling of *BB_sys.get_info*.

The manager process of Fig. 3.23 shows that the manager process of a root-BB-system behaves differently from the manager process of a child BB-system.
The cause of this difference is that all operations of *BB_sys* can be called from within the internal behaviours of KSs of a parent-BB-system. As a root BB-system has no parent-BB-system, the operation *BB_sys.create_BB_sys* of the root-BB-system has to be called from 'outside'.
BB_sys.modify_BB_sys, *BB_sys.get_info* and *BB_sys.finish_BB_sys* can also be called by 'outside' as these operations can be called by a parent.

The fact that *BB_sys* can be a root- or a child-BB-system, combined with the fact that *BB_sys.modify_BB_sys* and *BB_sys.finish_BB_sys* can be called from within a parent-system or from within the same system, complicates the manager process of *BB_sys* very much.

Note that for every state transition of the external behaviour of *BB_sys*, two transitions are needed in the manager process of *BB_sys*. This difference is caused by the switching between the subprocesses.

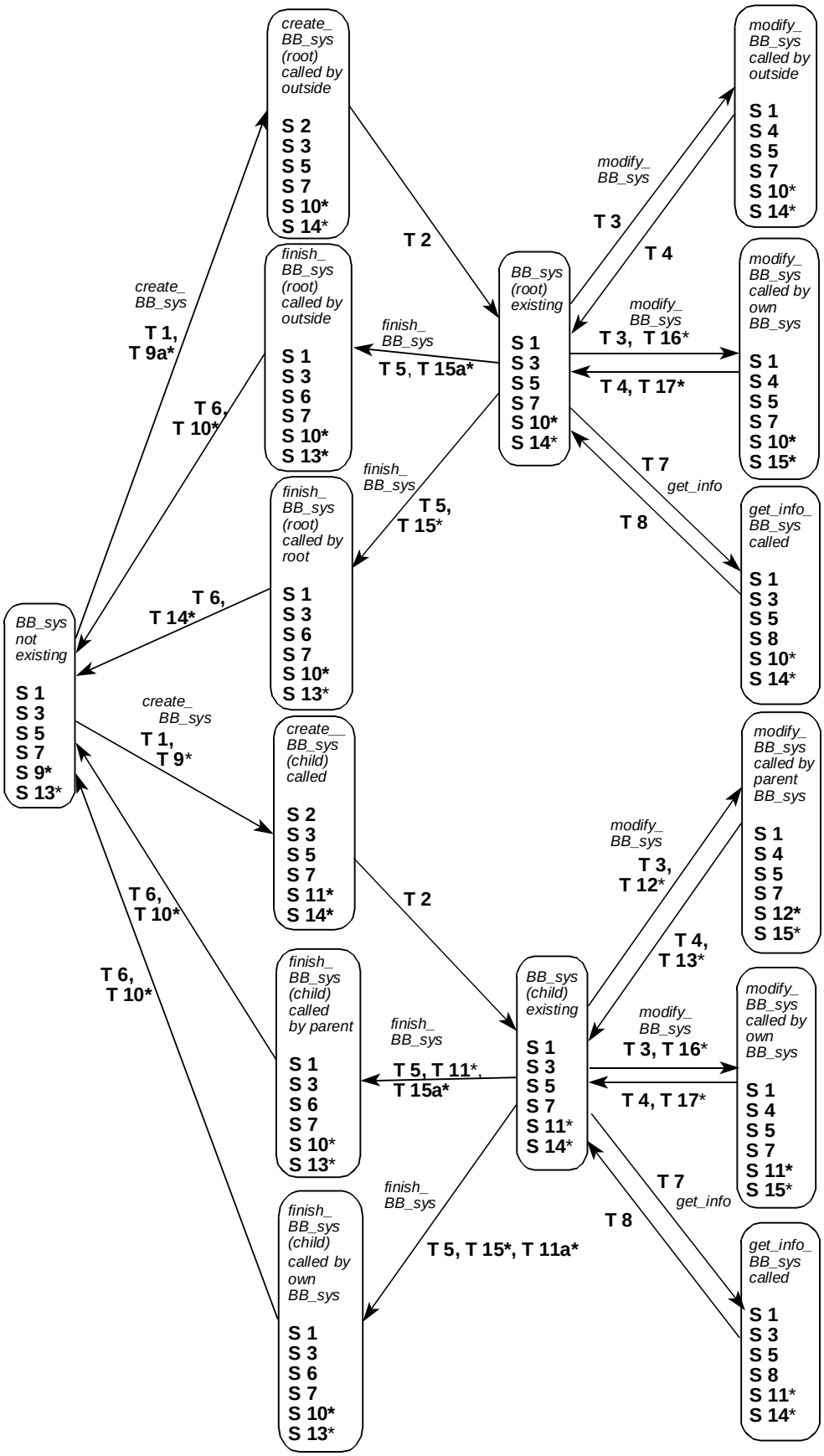


Fig. 3.23. *BB_sys*, manager of 6 employees
* : only the subprocesses of one KS per BB-system are drawn in the manager process

3.6.2. Subprocesses with respect to Knowledge Source (KS)

The manager process of *KS* is presented in Fig. 3.31.
Every KS object represents one role. So, for each role there is a separate manager process.

Subprocesses and traps in connection with the activation of the operations of *KS*:

In Fig. 3.24 the subprocesses and traps of *int-KS.activate_KS* with respect to *KS* are given.
A KS remains active until deactivation, so, the operation *KS.activate_KS* cannot terminate before the operation *KS.deactivate_KS* is called.
As soon as *KS.activate_KS* is activated, the internal behaviour of this operation will remain in trap T 2 until *KS.deactivate_KS* is called. When *deactivate_KS* is called, the transit from S 2 to S 1 can be made.
Note that subprocess S1 does not admit any new proposal selection or creation, the behaviour of *KS.activate_KS* can only terminate in S 1.

The subprocesses and traps (Fig. 3.25) of *int-KS.activate_proposal* are chosen as large as possible as the calls for *BB_sys.create_BB_sys, BB_sys.finish_BB_sys, BB_sys.modify_BB_sys* and *BB.modify_BB_sys* are already controlled by respectively *BB_sys* and *BB.*

The subprocesses and traps of *int-KS.deactivate_KS* (Fig. 3.26) are chosen so that the operation has to be terminated before a new operation can be handled by the manager process of *KS.*

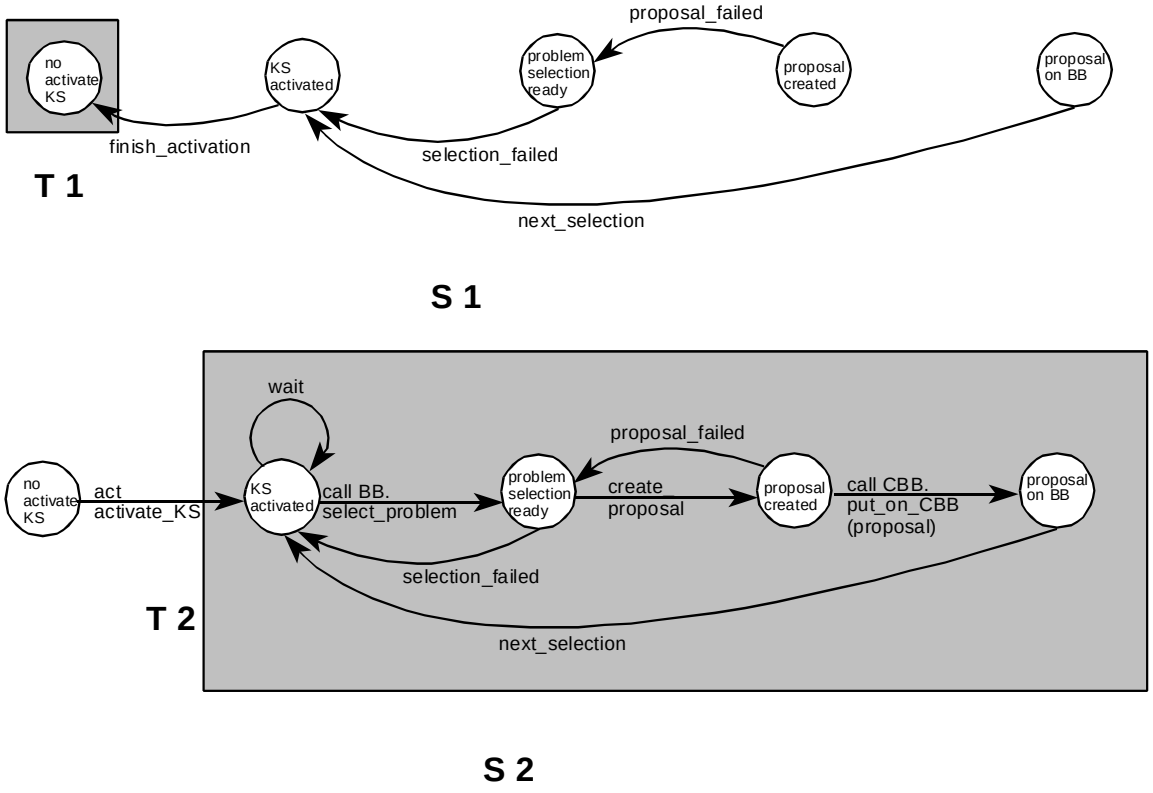


Fig. 3.24. S 1 and S 2 : subprocesses of *int-activate_KS* with respect to *KS*

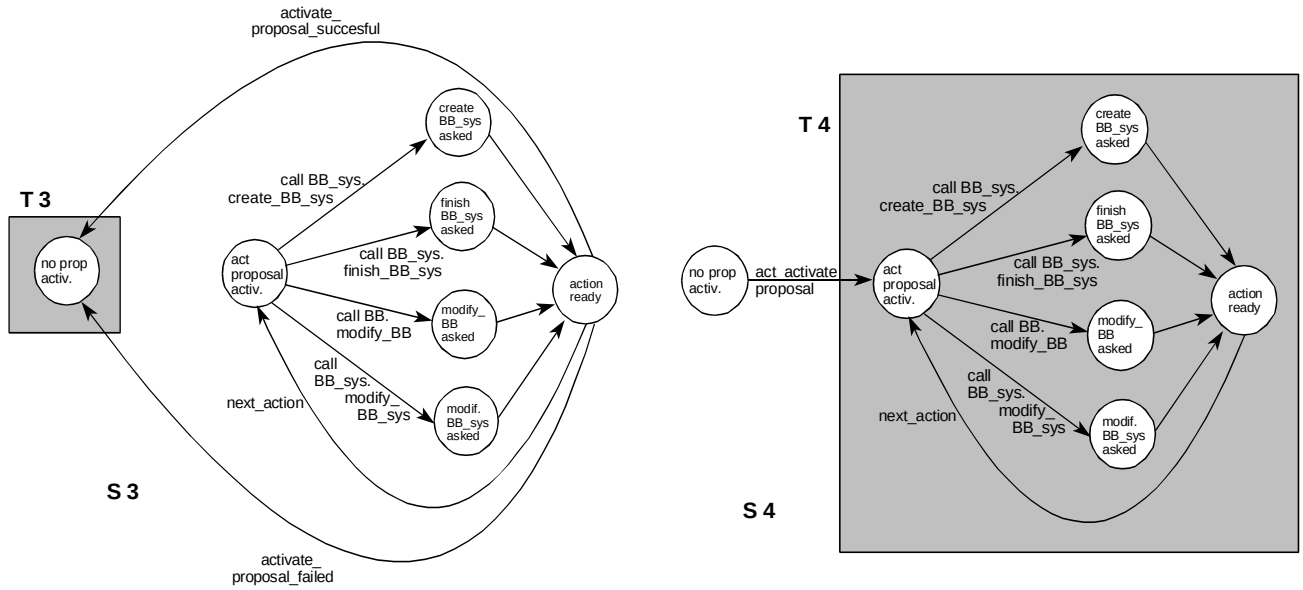


Fig. 3.25. S 3 and S 4 : subprocesses of *int-KS.activate_proposal* with respect to *KS*

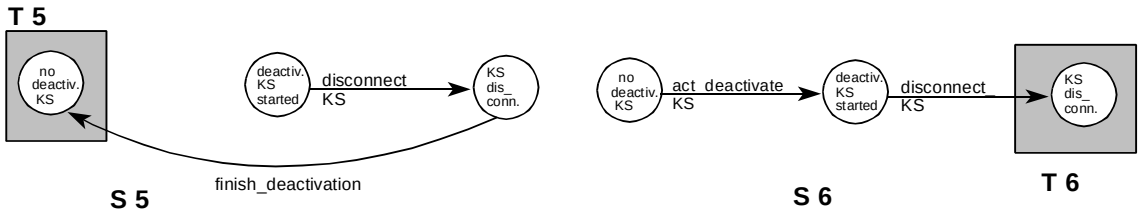


Fig. 3.26. S 5 and S 6 : subprocesses of *int-KS.deactivate_KS* with respect to *KS*

Subprocesses and traps in connection with the calling of the operations of *KS*:

The subprocesses and traps with respect to the calling of *KS.activate_KS* are given in Fig. 3.27. and Fig. 3.30. The trap T 8 contains the complete STD of *BB_sys.create_BB_sys* as the KS can be activated only once in connection with the same BB_sys. T 17 is chosen as large as possible so that the manager process of *KS* can admit another call this operation again as soon as possible.
Note that S 7 has an extra state 'activ. KS started', as other KSs may be activated before KS is to be activated

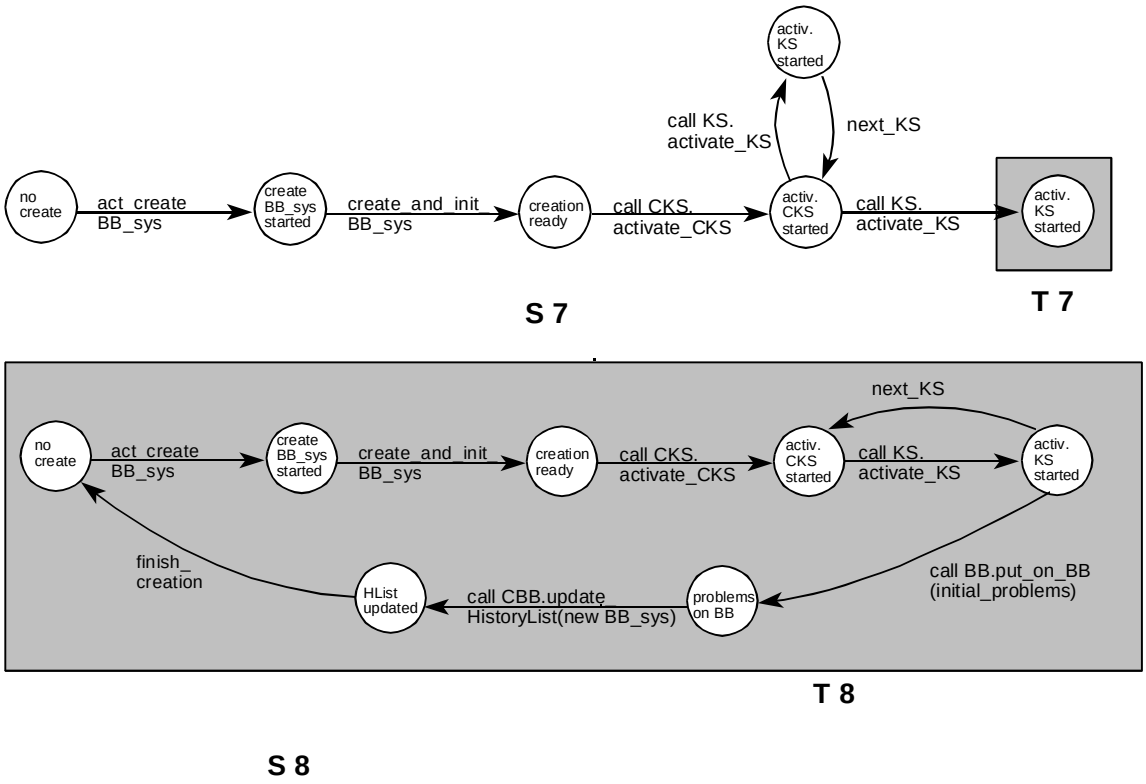


Fig. 3.27. S 7 and S 8 : subprocesses of *int-BB_sys.create_BB_sys* with respect to *KS*

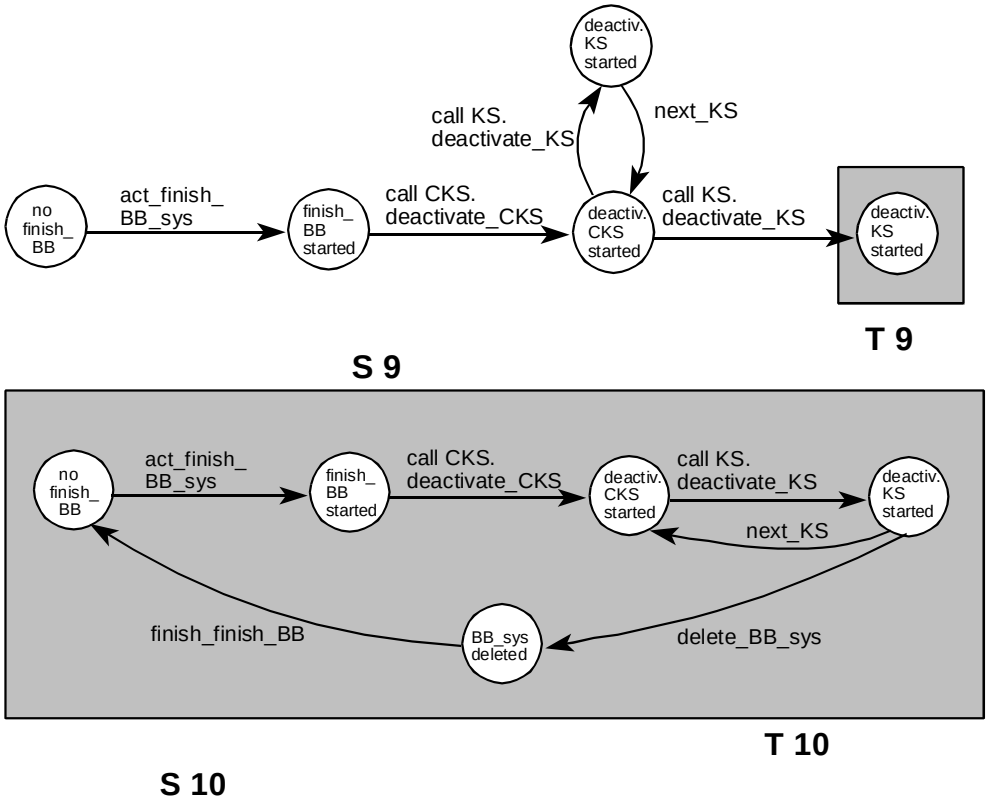
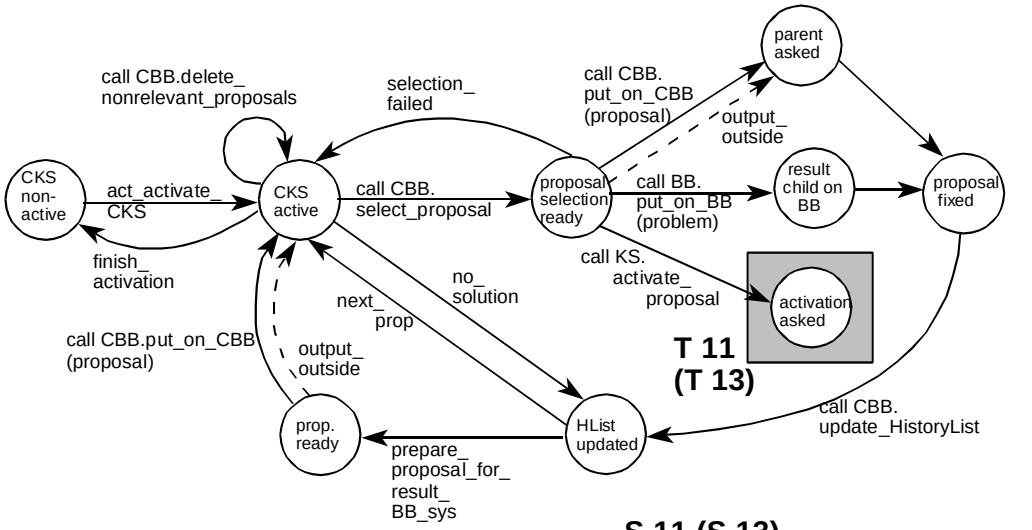
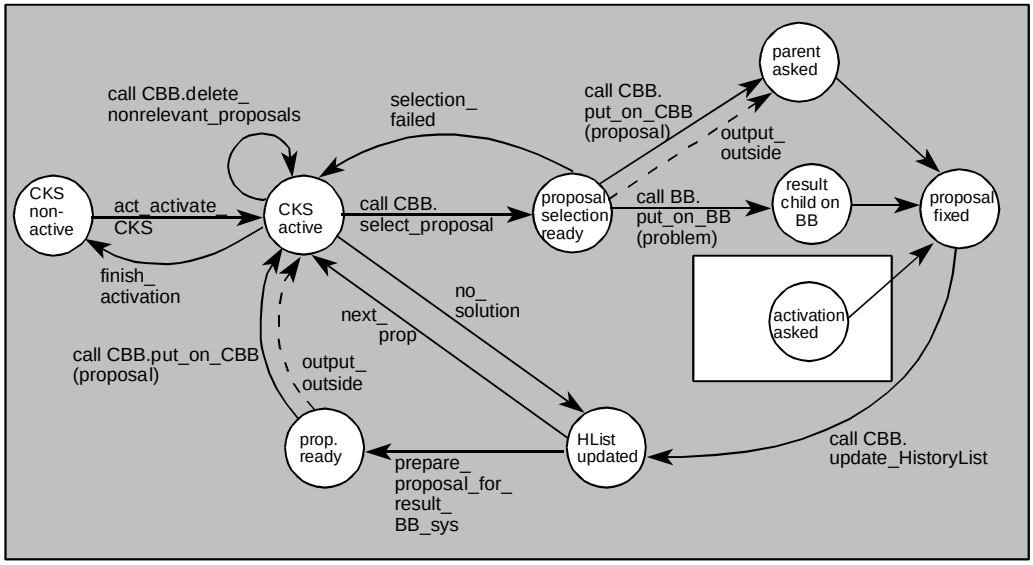


Fig. 3.28. S 9 and S 10 : subprocesses of *int-BB_sys.finish_BB_sys* with respect to *KS*



T 12 (T 14)



S 12 (S 14)

Fig. 3.29. S 11 and S 12 : subprocesses of *int-CKS.activate_CKS* in connection with the calling of *KS.activate_proposal* by the CKS of *BB_sys.*
S 13, S 14 and T 13 and T 14 are subprocesses and traps of *int-CKS.activate_CKS* in connection with the calling of *KS.activate_proposal* by the parent-CKS. They are exactly the same as resp. S 11, S 12 and T 11 and T 12.

The subprocesses and traps with respect to the calling of *KS.deactivate_KS* are given in Fig. 3.28. and Fig. 3.30. The trap T 10 contains the complete STD of *BB_sys.finish_BB_sys* as a *KS* can only be deactivated once in connection with the finishing of *BB_sys*.

S 11 and S 12 present the subprocesses of *int-CKS.activate_CKS* in connection with the calling of *KS.activate_proposal* by the CKS of *BB_sys.*

As *activate_proposal* can be also called by the CKS of a parent-BB-system, extra subprocesses are needed to handle these calls.
The subprocesses and traps that handle the calls of the CKS of the parent-BB-system, S 13, S14, T 13 and T 14, will be exactly the same as resp. S 11, S 12, T 11 and T 12.

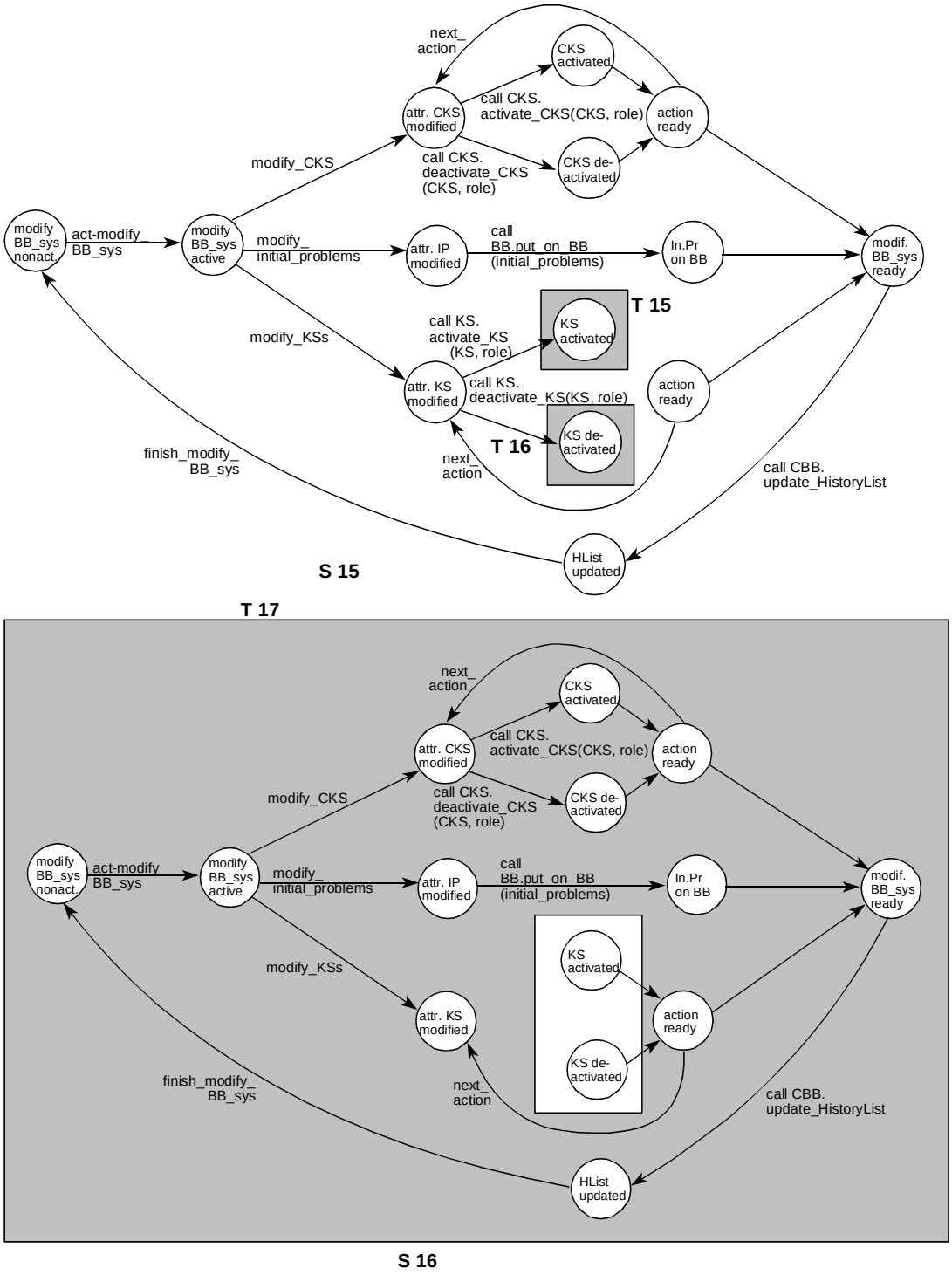


Fig. 3.30. S 15 and S 16: subprocesses of *int-BB_sys.modify_BB_sys* with respect to *KS*

The manager process of KS (Fig. 3.31) shows that the behaviour of KS is also affected by the type of the BB-system (root or child) the KS belongs to, as *KS.activate_proposal* can also be called by 'outside'

The manager process also shows that KS can only transit from S 2 to S 1 when the call for *KS.deactivate_KS* is made.

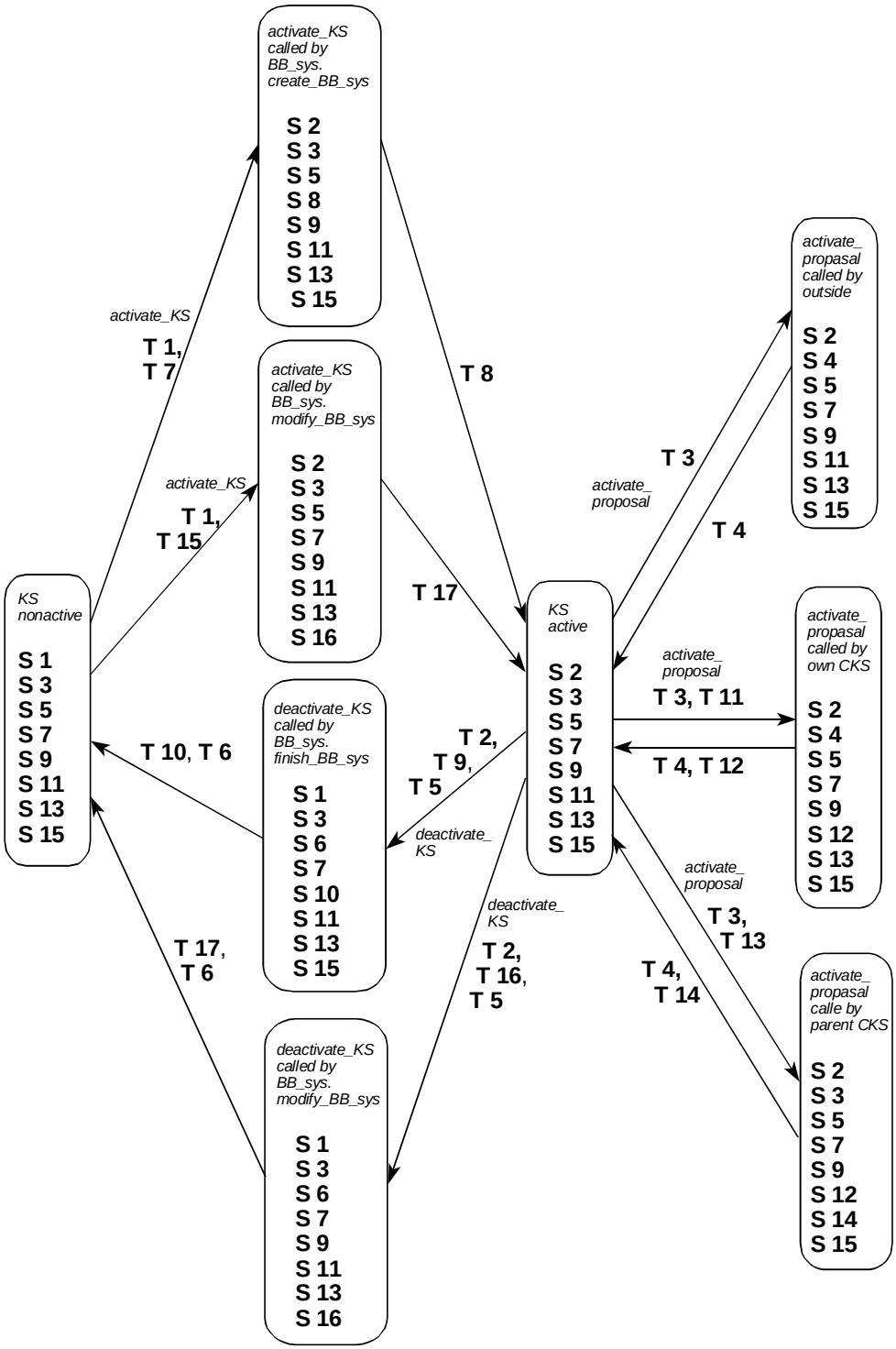
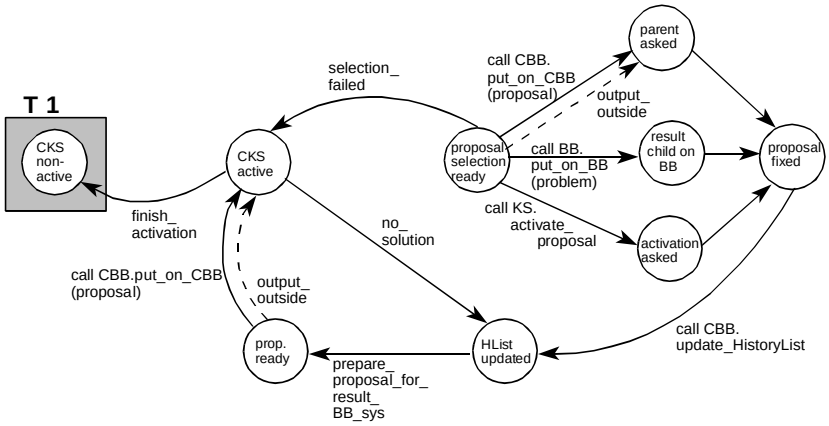


Fig. 3.31. *KS*, manager of 8 employees

3.6.3. Subprocesses with respect to Control Knowledge Source (CKS)

The manager process of *CKS* is very similar to the manager process of *KS*
If we leave out the operation *KS.activate_proposal* and its subprocesses in the manager process of *KS* , the remaining manager process is exactly the same as the manager process of *CKS*. The only thing left to do is to change 'KS' into 'CKS'. The manager process of CKS is given in Fig. 3.33.
The subprocesses S3, S4, S11, S 12, S 13 and S 14 of *KS.activate_proposal* are excluded from the manager process of CKS.
Like the KS, the CKS remains active until deactivation.

Only the subprocesses of *CKS.activate_CKS* will be presented, as all other behaviours of *CKS* are handled exactly the same way as the behaviours of *KS*.
The subprocesses and traps of *int-activate_CKS* are presented in Fig. 3.32.
Subprocess S 1 forces the operation *KS.activate_KS* to terminate as soon as possible, but also gives the CKS the opportunity to settle the already started actions. For instance, if the BB-system has finally come to a solution of the initial problems, the CKS can still bring the proposal for the result to the CBB of the parent-BB-system.



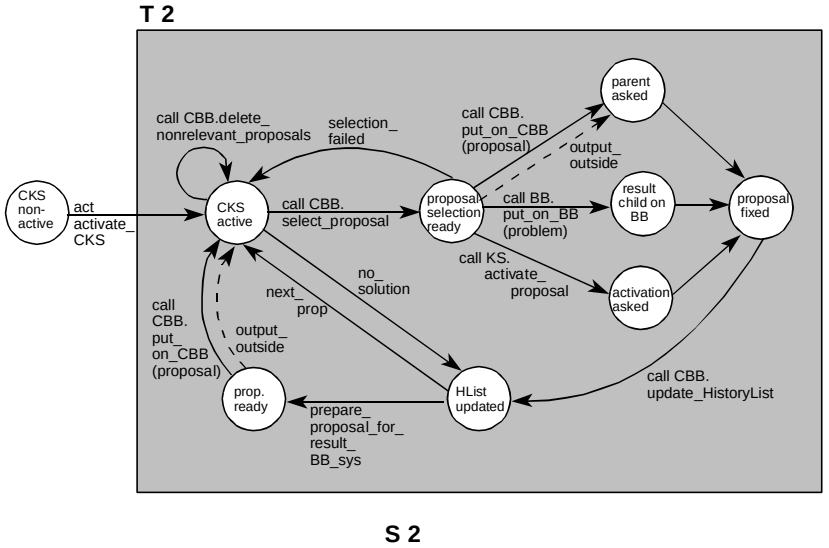


Fig. 3.32. S 1 and S 2 : subprocesses of *int-activate_CKS* with respect to *CKS*

The subprocesses and traps of *int-CKS.deactivate_CKS* (S 5, S 6, T 5, T 6) and *int-modify_BB_sys*

(S 15, S 16) are similar to the subprocesses presented in Fig. 3.26 and 3.30 of *KS*.
The subprocesses and traps of *int_create_BB_sys* (S 7, S 8, T 7, T 8) and *int-finish_BB_sys* (S 9, S 10, T 9, T 10) are also very similar to the subprocesses presented in Fig. 3.27 and 3.28, only the traps have to be modified to handle the calls for *CKS.activate_CKS* and *CKS.deactivate_CKS*.

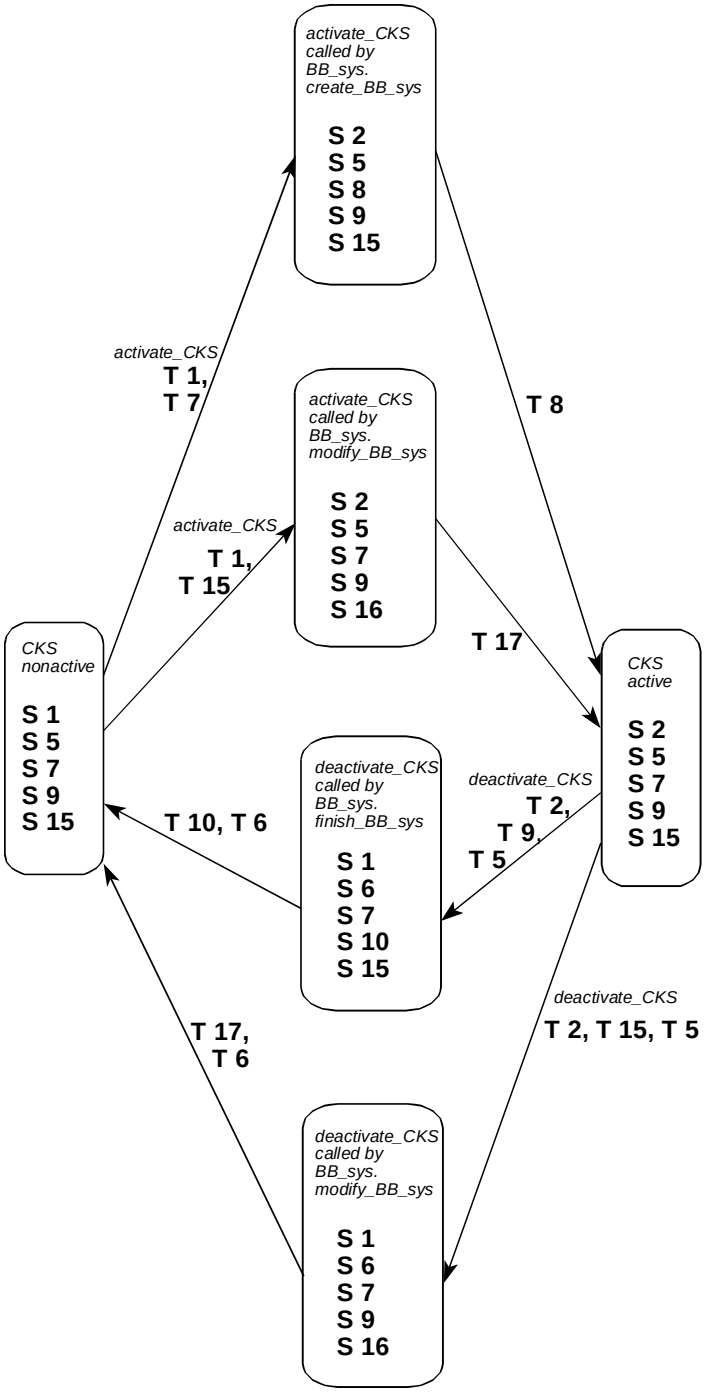


Fig. 3.33. *CKS*, manager of 5 employees

3.6.4. Subprocesses with respect to Blackboard (BB)

In Fig. 3.34. The manager process of *BB* is presented.
In section 3.6.1., the two ways of choosing traps in connection with the activation of an export operation are already explained.
The way subprocesses and traps are drawn with respect to the calling of an operation are also chosen in a standard way:

> (C1) If the called operation can be called again later in connection with the same instance of the class, the subprocesses and traps are drawn as S 11 and S 12 of *KS* as presented in Fig. 3.29:
> a subprocess with a trap containing the state immediately following the call for the operation and
> a subprocess with a trap that contains the complete STD of the operation except for the state from which the subprocess starts.
>
> (C2) If the called operation cannot be called again later in connection with the same instance of the class, the subprocesses and traps are draw as S 9 and S 10 of *BB_sys* as presented in Fig. 3.22:
> a subprocess with a trap containing the state immediately following the call for the operation and
> a subprocess with a trap that contains the complete STD.

As the traps of *BB* and *CBB* are always chosen in this standard way, the STD's of subprocesses and traps in connection with the activation and the calling of *BB* and *CBB* are omitted.
For every operation of *BB*, the subprocesses and traps will only be described

Subprocesses and traps in connection with the activation of the operations of *BB*:

BB.select_problem : subprocesses S 1 and S 2 with traps T 1 and T 2.
T 2 contains the final state before the finishing of *int-BB.select_problem* as caller *int-CKS.activate_CKS* can only continue when the result of *BB.select_problem* is known.

BB.modify_BB : subprocesses S 3 and S 4 with traps T 3 and T 4
T 4 contains the final state before the finishing of *BB.modify_BB* .
BB.modify_BB is called by *KS.activate_proposal*, that can transit to another subprocess before the operation is completely terminated (Fig. 3.25).
As caller *int-CKS.activate_CKS* has to be able to put the result on the CBB of the parent when the BB-system is declared solved or unsolvable, the operation *BB.modify_BB* has to be ready before the operation *CBB.put_on_CBB* (from the state 'prop. ready') can be called.

BB.put_on_BB : subprocesses S 5 and S 6 with traps T 5 and T 6
T 6 is chosen as large as possible.

Subprocesses and traps in connection with the calling of the operations of *BB*:

BB.select_problem : subprocesses S 7 and S 8 with traps T 7 and T 8.
BB.select_problem is called by *KS.activate_KS*.
T 8 is chosen as described in (C1) as *BB.select_problem* can be called again in *KS.activate_KS* in connection with the same BB.

BB.modify_BB : subprocesses S 9 and S 10 with traps T 9 and T 10.

BB.modify_BB is called by *KS.activate_proposal*.
T 10 is chosen as described in (C1) as *BB.modify_BB* can be called again in *KS.activate_proposal* in connection with the same BB.

BB.put_on_BB : subprocesses S 11 and S 12 with traps T 11 and T 12.
BB.put_on_BB is called by *BB_sys.create_BB_sys*.
T 12 is chosen as described in (C2) as *BB.put_on_BB* can only be called once in connection with the creation of the BB-system.

BB.put_on_BB : subprocesses S 13 and S 14 with traps T 13 and T 14.
BB.put_on_BB is called by *BB_sys.modify_BB_sys*.
T 14 is chosen as described in (C1) as *BB.put_on_BB* can be called again in connection with the modification of the BB-system.

BB.put_on_BB : subprocesses S 15 and S 16 with traps T 15 and T 16.
BB.put_on_BB is called by *CKS.activate_CKS*.
T 12 is chosen as described in (C1) as *BB.put_on_BB* can only be called again in *CKS.activate_CKS*.

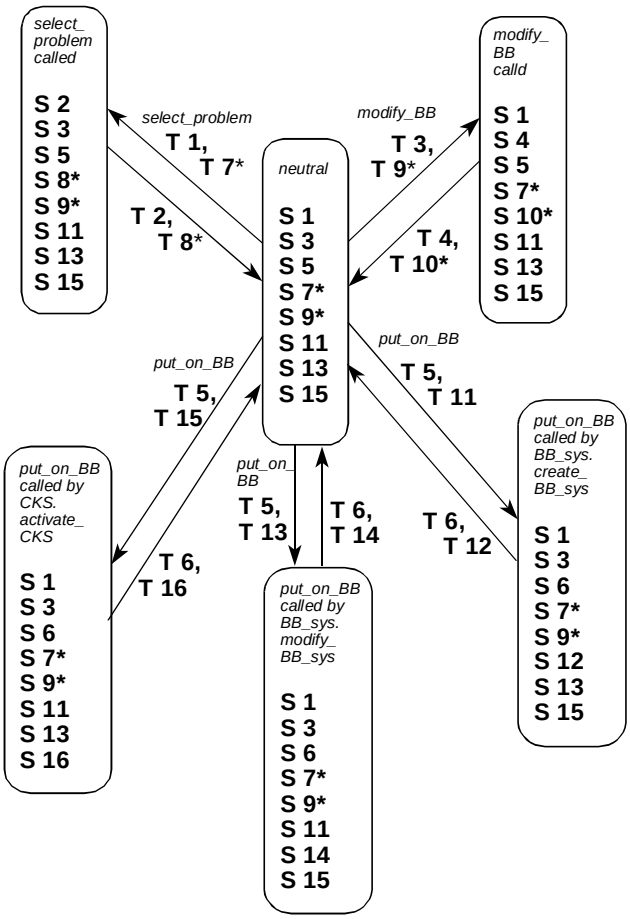


Fig. 3.34. *BB*, manager of 8 employees
* : only the subprocesses of one KS per BB-system are drawn in the manager process

3.6.5. Subprocesses with respect to Control Blackboard (CBB)

The manager process of *CBB* (Fig. 3.37.) is similar to the manager process of *BB.*

Subprocesses and traps in connection with the activation of the operations of *CBB*:

CBB.select_proposal : subprocesses S 1and S 2 and traps T 1 and T 2.
T 2 contains the final state before the finishing of *int-CBB.select_proposal* as *int-KS.activate_KS* can only continue when the result of *CBB.select_proposal* is known.

CBB.delete_nonrelevant_proposals : subprocesses S 3 and S 4 and traps T 3 and T 4.
T 4 contains the final state before the finishing of *int-CBB.delete_nonrelevant_proposals* to prevent that *int-CKS.activate_CKS* can select a proposal that is going to be deleted by *CBB.delete_nonrelevant_proposals.*

CBB.update_HistoryList : subprocesses S 5 and S 6 and traps T 5 and T 6.
T 6 is chosen as large as possible.

CBB.put_on_CBB : subprocesses S 7 and S 8 and traps T 7 and T 8.
T 8 is chosen as large as possible.

Subprocesses and traps in connection with the calling of the operations of *CBB*:

CBB.update_HistoryList : subprocesses S 9 and S 10 and traps T 9 and T 10.
CBB.update_HistoryList is called by *BB_sys.create_BB_sys*.
T 10 is chosen as described in (C2) as *CBB.update_HistoryList* can only be called once in connection with the creation of a BB-system

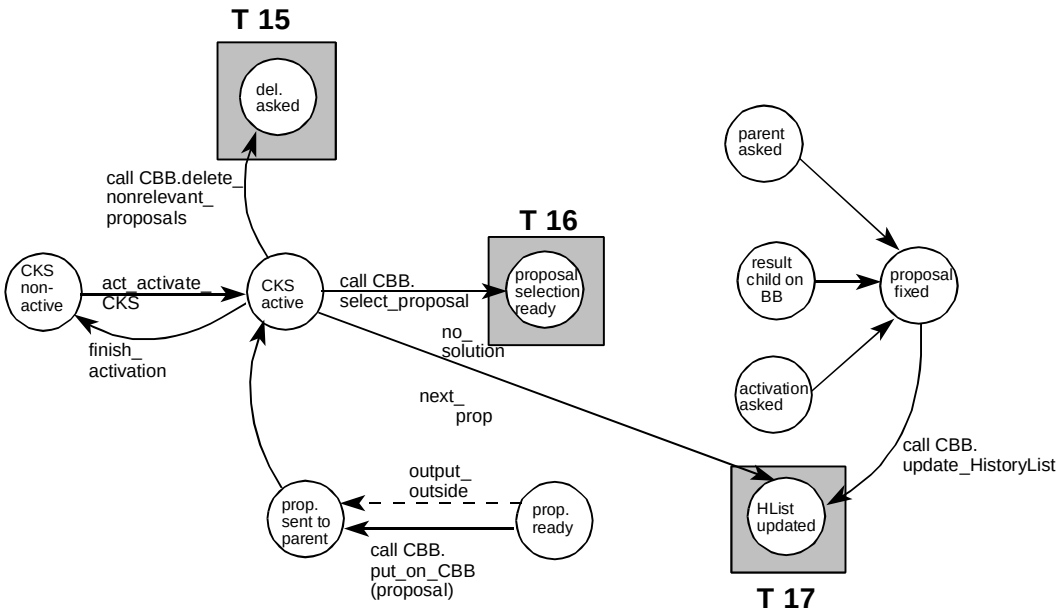
CBB.update_HistoryList : subprocesses S 11 and S 12 and traps T 11 and T 12.
CBB.update_HistoryList is called by *BB_sys.modify_BB_sys*.
T 12 is chosen as described in (C1) as *CBB.update_HistoryList* can be called more than once in connection with the modification of a BB-system

CBB.put_on_CBB : subprocesses S 13 and S 14 and traps T 13 and T 14.
CBB.put_on_CBB is called by *KS.activate_KS*.
T 14 is chosen as described in (C1) as *CBB.put_on_CBB* can be called more than once by the KS in connection with the same CBB.

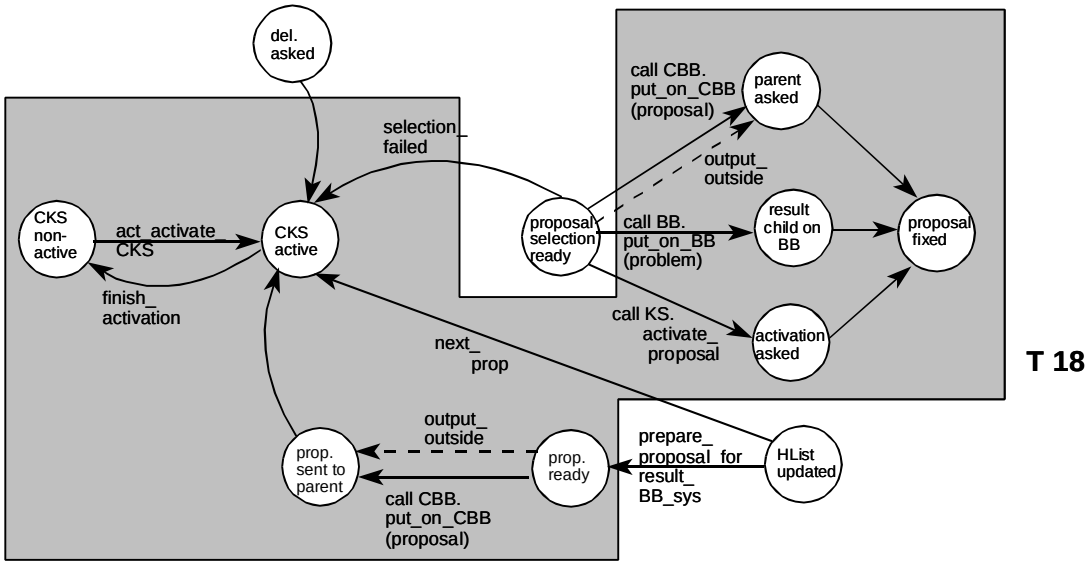
All other calls for operations of *CBB* are made from the internal behaviour of *CKS.activate_CKS*. Note that S 15 allows the calling of 3 different services and that S 17 allows the calling of 2 different services.

Fig 3.35 presents the subprocesses S 15 and S 16 with respect to the calling of *CBB.select_proposal*, *CBB.delete_nonrelevant_proposals* and *CBB.update_HistoryList* by the CKS of the same BB-system..

Fig. 3.36 presents the subprocesses S 17 and S 18 with respect to the calling of *CBB.put_on_CBB* by the CKS of the parent-BB-system.

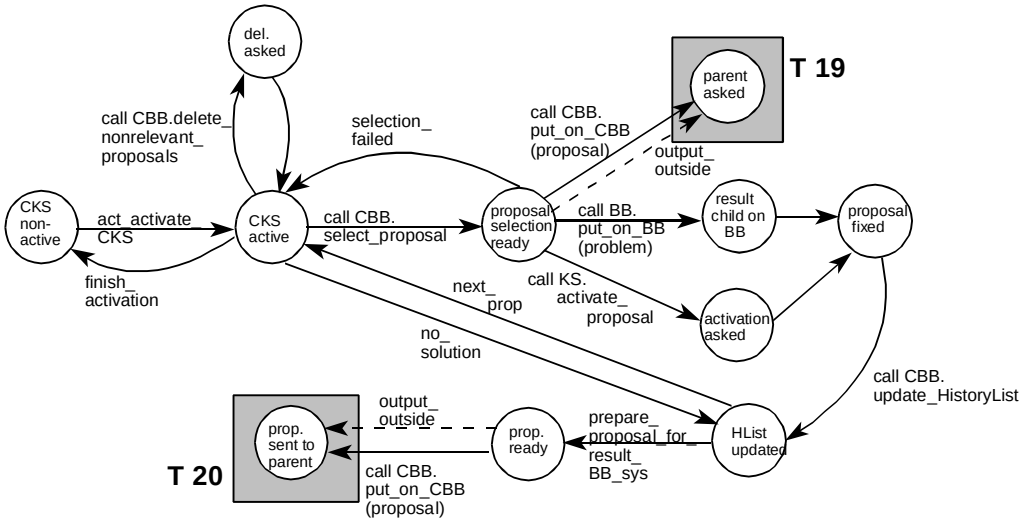


S 15

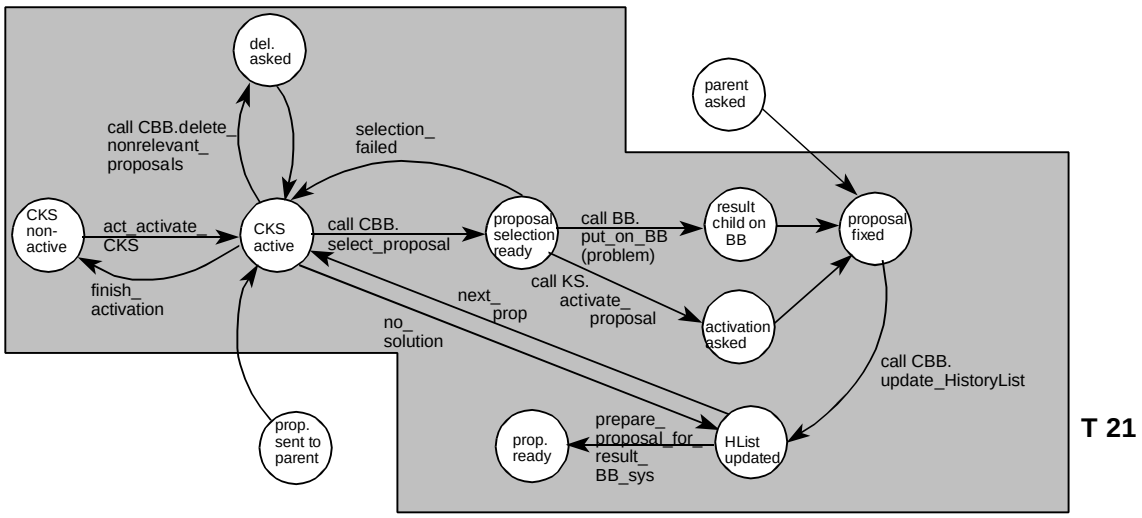


S 16

Fig. 3.35. S 15 and S 16 : subprocesses of *int-activate_CKS* with respect to the calling of *CBB.select_proposal*, *CBB.delete_nonrelevant_proposals* and *CBB.update_HistoryList* by the CKS of the same BB-system.



S 17



S 18

Fig. 3.36. S 17 and S 18: subprocess of *int-activate_CKS* with respect to the calling of *CBB.put_on_CBB* by the CKS of the parent-BB-system.

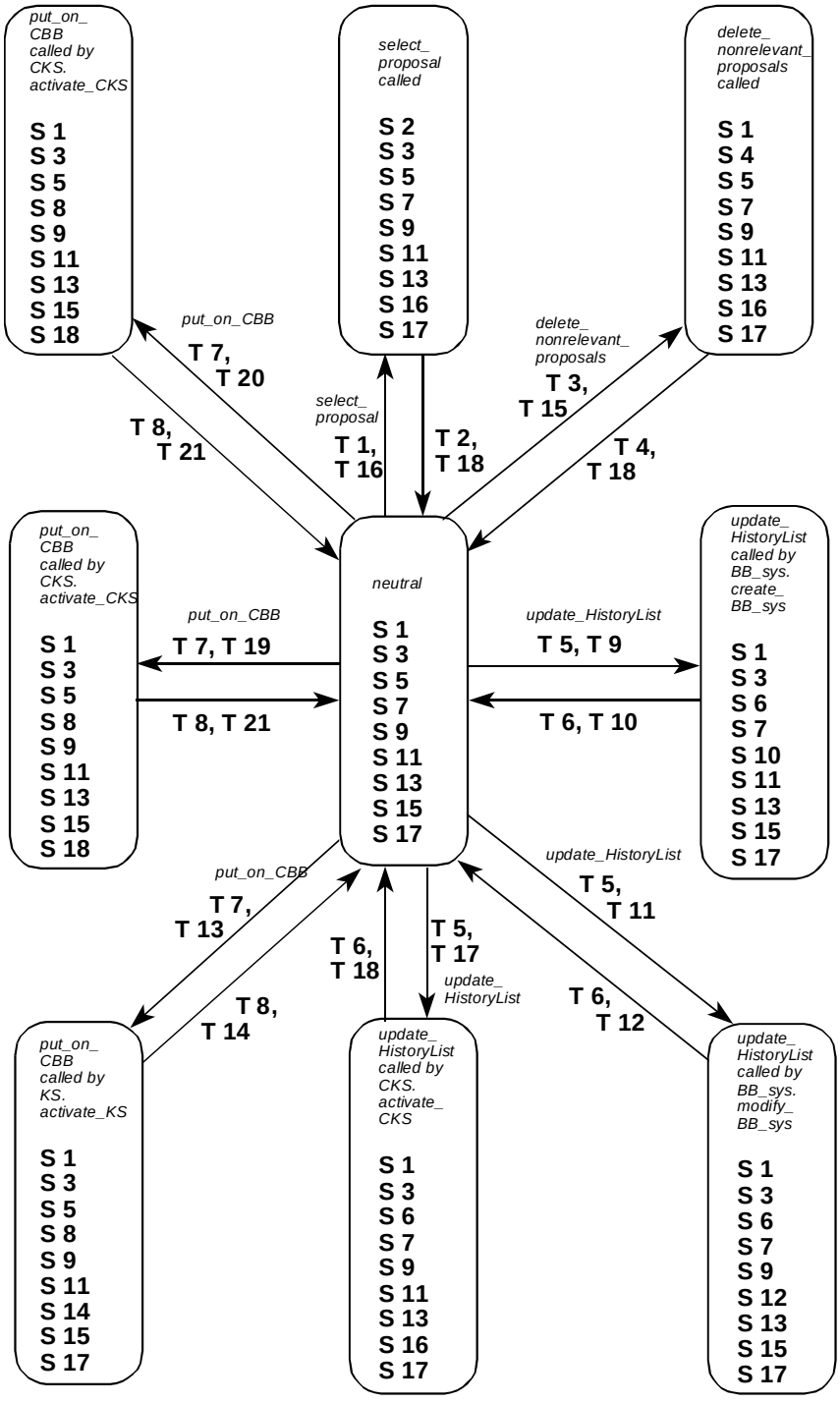


Fig. 3.37. *CBB*, manager of 9 employees

Part IV: Application of the given example

4.1. Introduction

In this part the SOCCA model is applied to the given example, as described in section 1.5.
Two different representation types will be used to illustrate the way the BB-systems process the details of the given example.
The actual calling of the export operations is worked out in event traces. In addition to these event traces, process models will represent the state of the BB-systems at fixed points of time.
The complete event trace is divided into 9 steps. Every step is concluded with the creation of a new process model. This process model, representing a BB-systems as presented in section 2.1.1, is shown for every step.

4.2. Event traces

Event traces are a well known common 'tool' for case-oriented analysis or specification of communication triggering. We will use this method to illustrate how the SOCCA-model for a Blackboard System can handle the writing of a book, as described in the given example.
The details of the given example will be translated into the parameters of the operations.

To simplify the event trace, it is not possible to show parallel communication between objects. Only one 'possible' sequence of events will be shown.

For the sake of 'readability' of the event trace, most proposals for an action are followed immediately by their activation. This is not a very probable sequence in a working Blackboard System. But, if the actual sequence of proposal-creation and -activation of different proposals does not influence the problem-solving activity, we will maintain this order of events.

For the same reason, all proposals will be selected and activated by the CKS. A proposal that is rejected does not show any action, it will only be deleted.

4.3. The export operations and their parameters

The parameters of the operations are given in fig. 4.1.
The given parameters do not represent the actual implementation details. Some parameters are only shown for the sake of readability.

The parameter *ok/not_ok* is a boolean parameter of the operations *CBB.select_proposal* and *BB.select_problem*. The returned value indicates whether the selection was successful or not.
The parameter *caller* of the operations of *BB_sys* and *KS* denote the 'calling BB-system' or 'outside'.
The operation *BB.put_on_BB* can be called by *BB_sys* or by *CKS*. A CKS will use this operation to put a result of a child-BB-system on the BB. *BB_sys* will use this operation to put new initial problems on the BB. For this reason, the third parameter of *BB.put_on_BB* will be either *result* or *Initial_Problems*.

The details in the event trace, representing the parameter actions of the export operation *BB.modify_BB,* cannot be traced back in the STD's of the SOCCA-model. As all modifications on the BB are executed by internal operations of the internal behaviour of *BB.modify_BB,* the STD's do not specify these operations any further.

BB_sys
create_BB_sys (new-BB_sys, caller, Initial_Problems, KSs + Roles, CKS + Role)
modify_BB_sys (called-BB_sys, caller, modifications)
finish_BB_sys (called-BB_sys, caller)
get_info (called-BB_sys, caller)

KS
activate_KS (BB_sys, KS + Role)
activate_proposal (called-BB_sys, caller, Proposal, KS)
deactivate_KS (BB_sys, KS + Role)

CKS
activate_CKS (BB_sys, CKS + Role)
deactivate_CKS (BB_sys, CKS + Role)

BB
select_problem (BB_sys, ok/not_ok, Problem)
modify_BB (BB_sys , Problem, actions, Proposal)
put_on_BB (BB_sys, Problem, Initial_Problems/result)

CBB
select_proposal (BB_sys, ok/not_ok, Proposal)
put_on_CBB (called-BB_sys , caller-BB_sys, Proposal)
update_HistoryList (BB_sys, Problem, registration_History)
delete_nonrelevant_proposals (BB_sys, Proposals)

fig 4.1. Export operations and their parameters (in the context of the event trace)

4.4. The division of the example into BB-systems, child-BB-systems, problems and subproblems

The KSs communicate through the BB with other KSs by putting new subproblems on the BB.
The schema's of appendix A show the translation of the informal description of the given example into problems and subproblems, BB-systems and child-BB-systems.
Every step will introduce new problems and/or new BB-systems. The schema's of appendix A can help to keep track with all these problems and BB-systems.

The division into problems and subproblems must be viewed as only one possible interpretation of the given example. In connection with the example, innumerable other acceptable interpretations are possible.
The SOCCA model emphasises the communication between the BB-systems more than the problem-solving on the BB. For this reason, the selection of the subproblems concentrates on the identification of the separate BB-systems that model the processes of the given example. The number of subproblems is kept low to avoid too many details.
Unfortunately, this low number of subproblems complicates a 'realistic' representation of the details of the given example. However, on the basis of the process models and event traces presented in this section, it is not difficult to imagine a more realistic processing of the details of the example.

4.5. Representation of the example in 9 steps

4.5.1 Step 1: The creation of the root-BB-system *Process Creation*

In this section we will present the start-up of the activities of the given example.

To start up the process of the collaborative writing of a book, we will first create a root-Blackboard System that can create and activate metaprocess-like activities.
The root-BB-system will represent Jean-Claude, viewed from the perspective of the PROMOTER Community. Jean Claude is the coordinator of the PROMOTER community and we are especially interested in the role that he plays in the start-up and registration of the processes that are described in the example.

Fig. 4.2. shows the process model of this root-BB-system, named *Process Creation* . The model shows the two KSs and their roles: Jean Claude (JC) as 'Process creator' and JC as 'Process Model creator'.
JC is also the CKS of the BB-system, as he performs the role of 'co-ordinator'.
As discussed in section 2.5, a person can have more than one role at the same time.

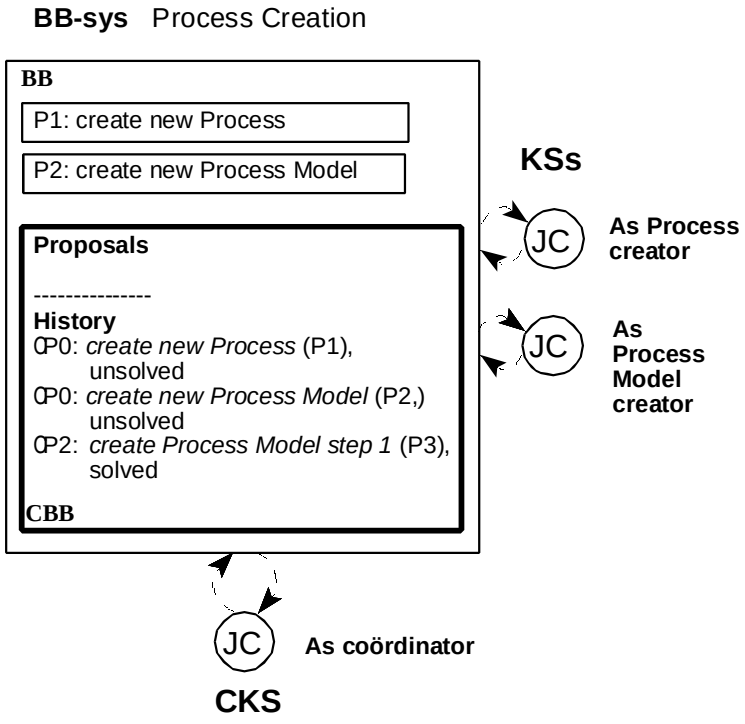


fig 4.2. Process model step 1

There are two problems on the BB : *create new Process* and *create new Process Model*. The first problem, *create new Process* can create and start-up any possible new process.
The solving of this problem can be viewed as a meta-activity: the problem can start-up new processes forever and remain unsolved as long as the KS Jean Claude as Process creator likes.
The same applies to the problem *create new Process Model*. This problem can register the evolving of the processes taking place in the example by creating a process model for every significant step.
The KS, Jean Claude as Process Model creator, will create a new process model after every step by creating and solving a new subproblem named *create Process Model Step x.*

The process model shows in the *HistoryList* on the CBB what action has taken place on the BB before the process model was created.
In the HistoryList we can also trace back the parent-problem of every problem. The parent-problem of the initial problems is P0, representing the parent-BB-system with the problem that caused the new BB_systems. The problem *create Process Model Step 1* is a sub problem of P2, *create Process Model.*
The HistoryList also shows that at the moment the process model is created, the two initial problems are unsolved and the problem P3, *create Process Model Step 1*, is solved.

To avoid too many details in the process model, only the unsolved problems will be shown on the BB of the process model. In addition to this, only the results (solutions or failures) of the child-BB_systems will be shown on the BB of the process model.
The details of all other solved or unsolvable problems can be found in the HistoryList of the CBB.

There are no proposals on the CBB. The process model was drawn when all proposals were activated and updated in the HistoryList.
Usually, there may be many proposals on the CBB. We will only show the proposals that will be activated later. As we want the event trace to show the activation immediately after the proposal of a new subproblem, there will hardly be any proposal on the CBB in our process models. Again, this not very likely to happen in a working Blackboard System.

Fig. 4.3 presents the event trace of step1.
Section (1) of the event trace shows the creation and activation of the very first Blackboard System, *Process Creation.*
The parameter 'outside' that denotes the caller of the operation *create_BB_sys* shows that the BB-system to be created is the very first or root-BB-system.
We can see that every KS and CKS involved is to be activated separately. The calls for all activate-operations are made from the internal behaviour of the operation *create_BB_sys.* As the dotted vertical lines indicate the operations of the classes, we can trace back for every call from which internal behaviour the call is made.

As soon as the problems are put on the BB, they can be selected by the KSs by calling *BB.select_problem*.

All remaining operations in the event trace of step 1 show the proposal for and activation of the creation of the new process model of step 1.
All process models that are created in a later stage, will be made by the calling of exactly the same operations and parameters.
For this reason, only an empty grey square will be shown when the next process model is being created.
The only change to the parameters of the operations is the name of the subproblem of the operation *create new process model*. The name of this subproblem is in step 1 : *create process model step 1*, in step 2: *create process model step 2*, etc.
The names of these operations can be found in the HistoryList, drawn in the process model.

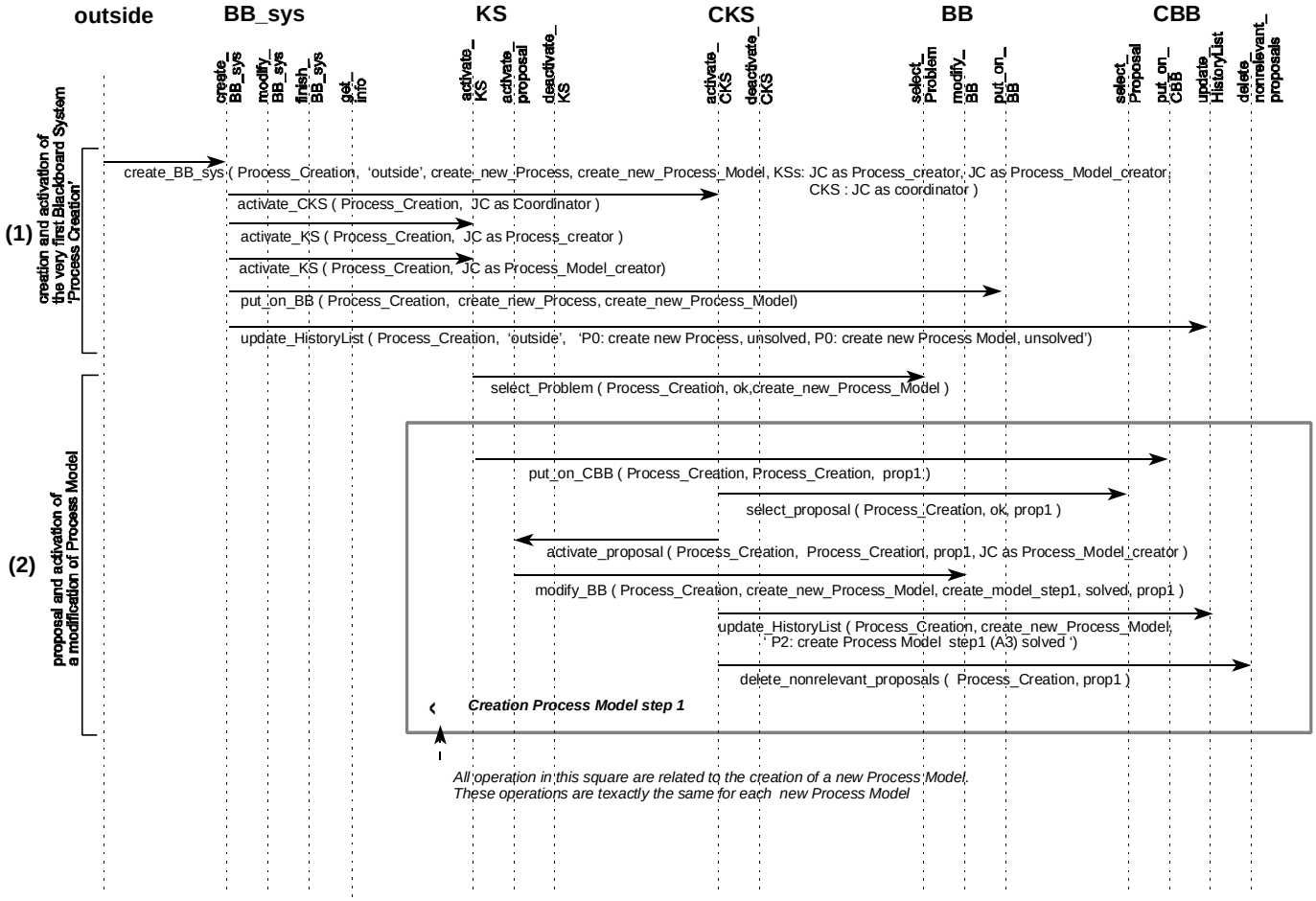


fig 4.3. Event trace step 1

4.5.2 Step 2: Creation and activation of the first child-BB-system *Promoter Meeting*

This step describes the starting of the Promoter meeting.

As the process model shows (Fig 4.4) , JC as process creator has proposed and activated a new process: the Promoter Meeting. The initial problem on the BB of *Promoter Meeting* is *go through agenda*. The formulation of the initial problem indicates that this problem is solved when the KSs have made decisions about all topics on the 'agenda'.
The 'agenda' will have to be defined in the problem description of *go through agenda*. However, like in usual meetings, a KS can also add a topic to the 'agenda'.

The KSs of *Promoter Meeting* are the members of the Promoter community and JC as the leading chairman. Jean Claude is the CKS as he performs the role of controlling chairman.
We need this refinement of the role of chairman as Jean Claude may also have to influence the meeting in a more 'active' way (see also 2.6).

The dotted arrow from *Process Creation* to *Promoter Meeting* shows that the BB-system *Promoter Meeting* is activated by the parent-BB-system *Process Creation*.
The arrow labelled 'info' indicates that the KS, JC as Process Model creator, asked for info about the new BB-sys in order to make the process model of step 2.

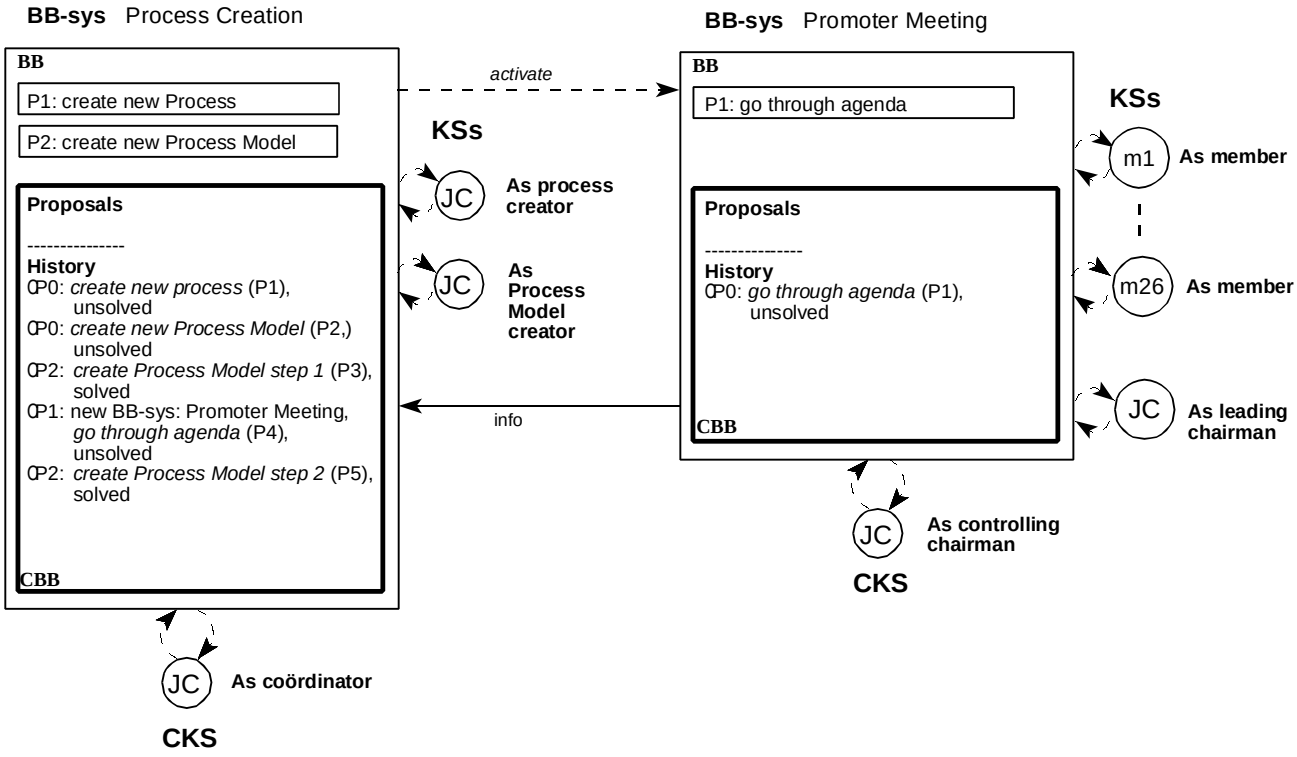


fig 4.4 . Process model step 2

Fig. 4.5. shows the event trace of step 2.
Section (3) of the event trace of step 2 shows the selection of a problem on the BB and the call for *put_on_CBB* to put the proposal on the CBB. The proposal concerns the creation of a new BB-system, named 'Promoter Meeting'.
In section (4), this proposal is accepted by the CKS. The KS that created the proposal is then asked to activate the proposal.
The new BB-system is activated and the HistoryList of both Blackboard Systems is updated.
This step ends with the creation of the process model of step 2.
Before JC can make the next model, he receives the most recent information about the new BB-system by using the operation *BB_sys.get_info*.
The creation of the new process model is described in section (5)

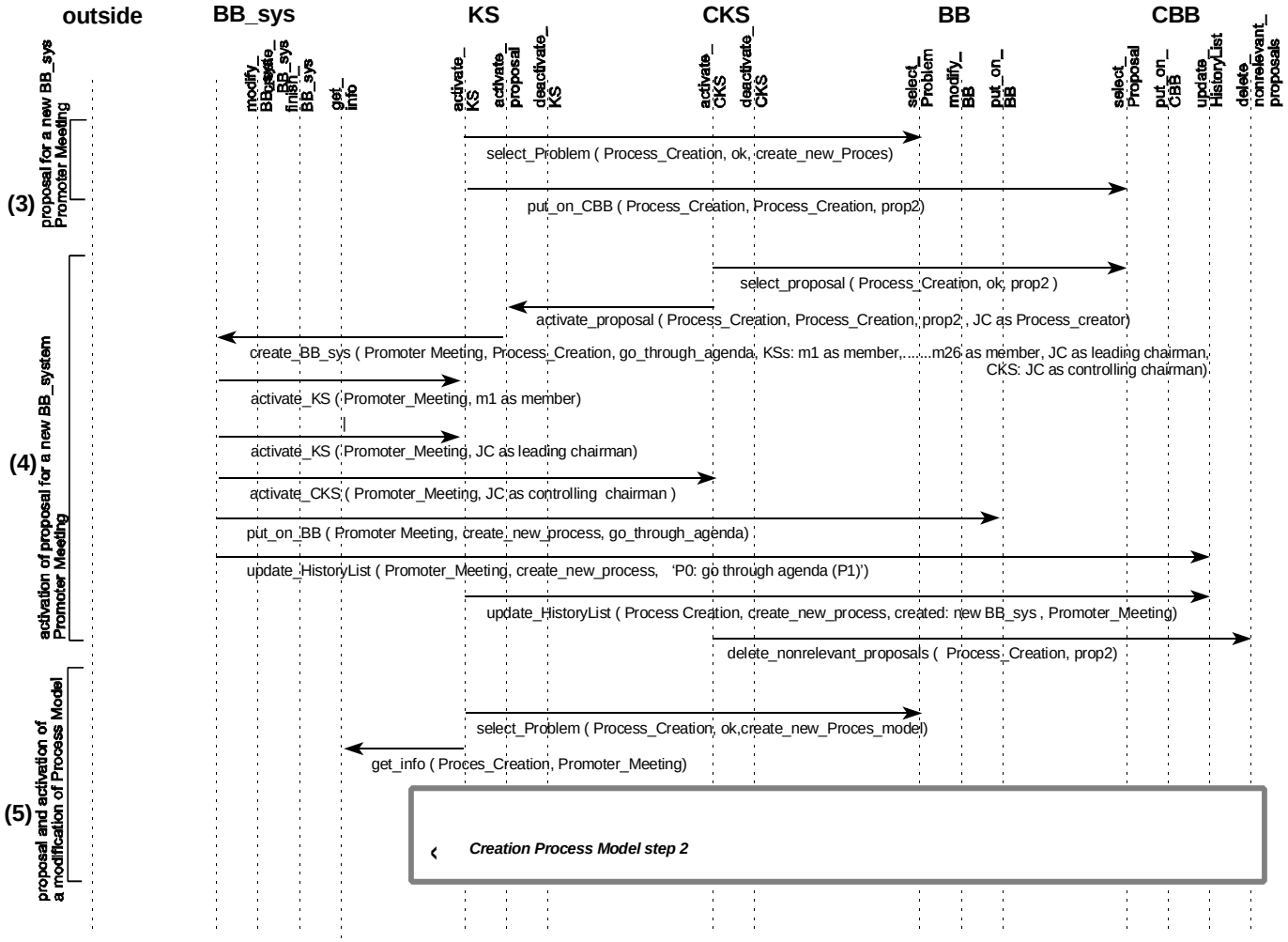


fig 4.5 Event trace step 2

4.5.3 Step 3: BB-system *Promoter Meeting* makes decisions about the second book

In step 3, the first decisions are made by the KSs of Promoter Meeting concerning the questions raised during the meeting of 19940209 as described in the given example..

The History on the CBB of *Promoter Meeting* shows all decisions made by the KSs of *Promoter Meeting*.
The first new problem, *make decisions about Book2*, is probably a topic on the 'agenda' and put on the BB by JC as leading chairman.
The first question raised by this new problem is whether the members of the Promoter community want to make this new Book. In the History on the CBB we can see that a subproblem named *make Book2?* is created by a KS with an immediate answer 'yes'. As this new subproblem has the state 'solved', we can say that the KS that created this subproblem 'posits' that a new book must be created.

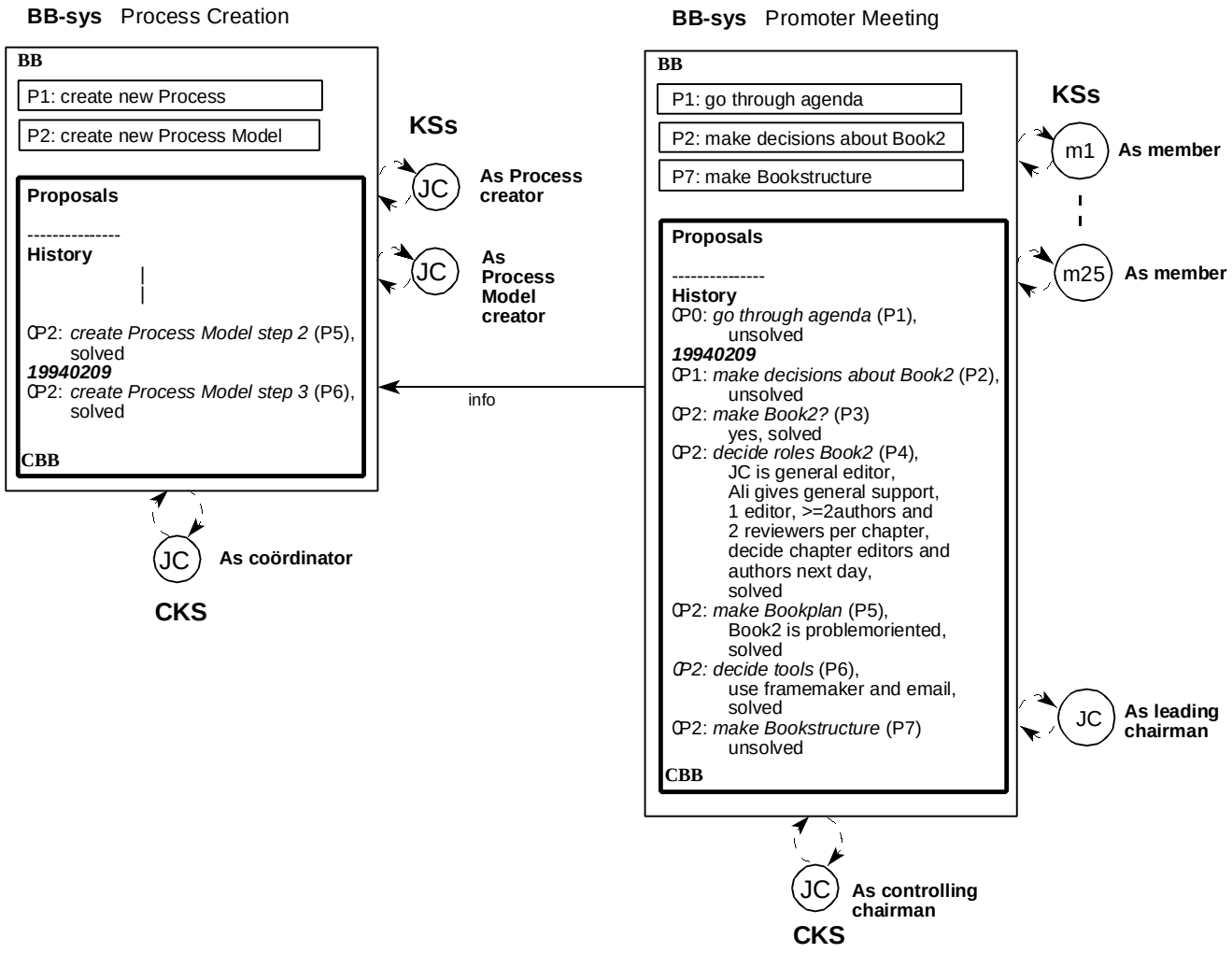


fig 4.6. Process model step 3

The decision to make a second book, in these steps referred to as Book2, is the first democratic decision made by the KSs of Promoter Meeting.
The SOCCA-model does not regulate this democratic decision making. The knowledge of the KSs and CKS has to define their behaviour in a democratic situation.
The KS Jean Claude as leading chairman has to find out whether a majority of the KSs agrees with a decision and the KSs have to make sure that their opinion concerning a decision on the BB is known by JC as leading and controlling chairman. The CKS has to control this democratic decision making. There are several ways to enforce democratic decision making on the BB.
To simplify the actions on the BB, we will assume that KSs only react to a decision on the BB if they do not agree completely with the decision and that the KS in the role of leading chairman gives the KSs time to react on a decision.

The given example only relates ‘vague’ details about the way decisions were made at the Promoter Meeting. As we are mainly interested in the ‘complete’ process of the collaborative creating of a book, we will not pay too much attention to the way KSs respond to new subproblems on the BB.

If we want to present a more 'realistic' way of decision making at the Promoter Meeting , we would need many more steps to model the example.
The problem *make decisions about Book2* causes many new subproblems like: *decide roles Book2, make bookplan* and *make Bookstructure*.
At the time the process model of step 3 was created, all new subproblems are solved, except *make decisions about Book2* and *make Bookstructure*.

In the given example, date descriptions are used to indicate at what point of time events took place. It is a very natural activity of the CBB to register the actual date of the actions on the BB.
As the given example only relates dates occasionally, only the dates mentioned will be processed in the History of the process models.

A modification of the BB can be represented by the following sequence of events:

select_problem();
put_on_CBB();
select_proposal();
activate_proposal();
modify_BB ();
update_HistoryList();
delete_nonrelevant_proposals();

As we confine ourselves to one sequence of events to simplify the event trace of the example, we only have to define the parameters of the listed operations that reflect the modification on the BB.
The given order of events is already illustrated in the event trace of step 1 (Fig 4.3). Note that there is one small difference with the events of the event trace of step 1: the updates in the sections (6) and (11) show the addition of an 'unsolved' subproblem to the BB instead of a 'solved' subproblem. However, the state of the added subproblem does not affect the sequence of events.

In the event trace of step 3 (Fig 4.7), all actions proposed and activated by the KSs are modifications of the BB.
As the interaction between the objects is already worked out in section (2) of the event trace of step 1, we will only specify the parameters of the operation *CBB.update_HistoryList* .
The parameters of *CBB.update_HistoryList* show all details of the proposed and activated modifications on the BB.
We will leave out all other operations of the event trace of step 3.
By doing this, some information will be lost, such as the actual KSs that proposed and activated the modifications of the BB. As the given example does not tell us who took the decisions in the Promoter Meeting of 19940209, we will just assume that the 'unsolved' subproblems are added by JC the leading chairman and the 'solved' subproblems by different members.

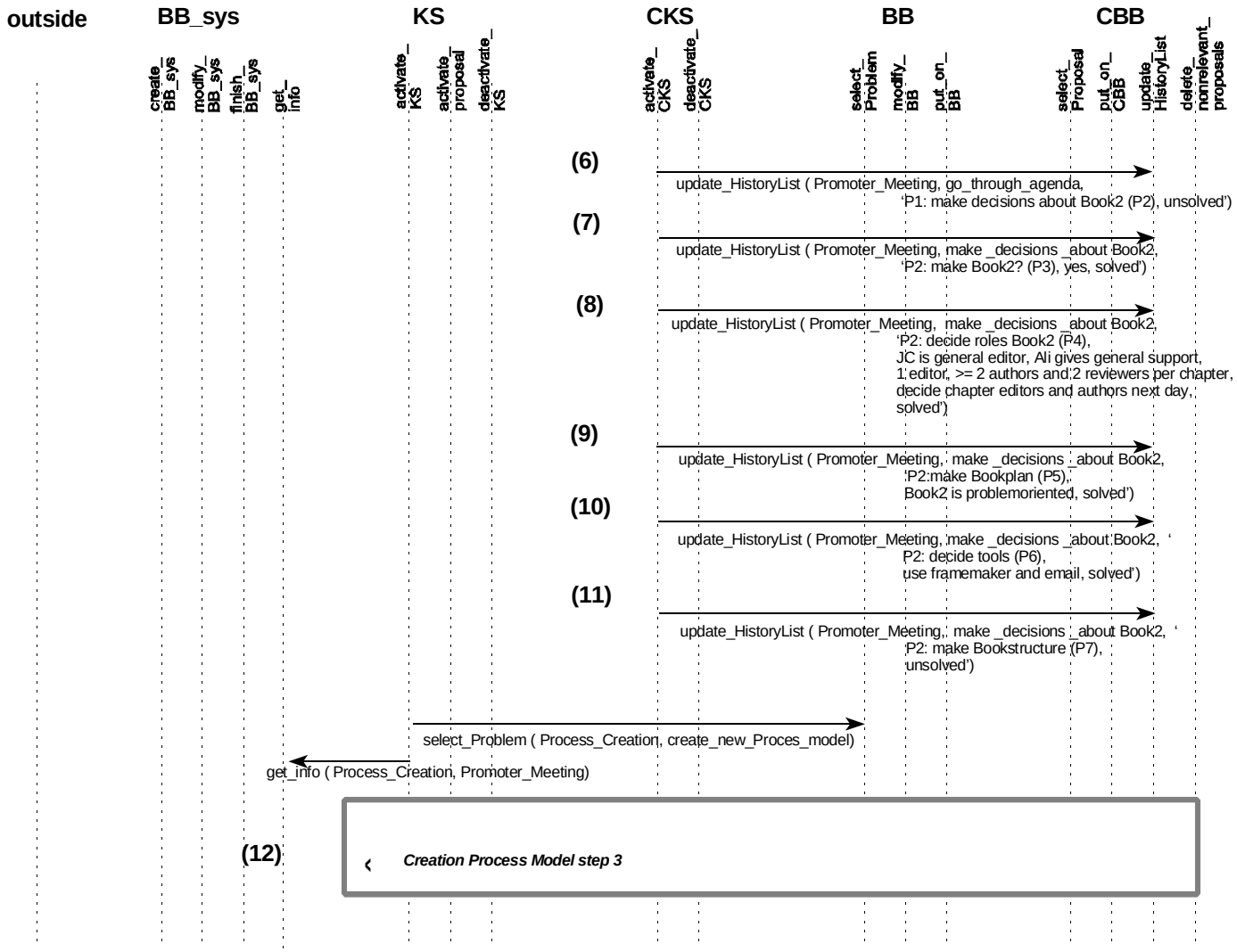


Fig 4.7 Event trace step 3

4.5.4 Step 4: The creation of more than one child-BB-system to solve a single problem.

The problem *make Bookstructure* remains unsolved in step 3. In step 4, the members are asked to prepare a bookstructure for the following day.

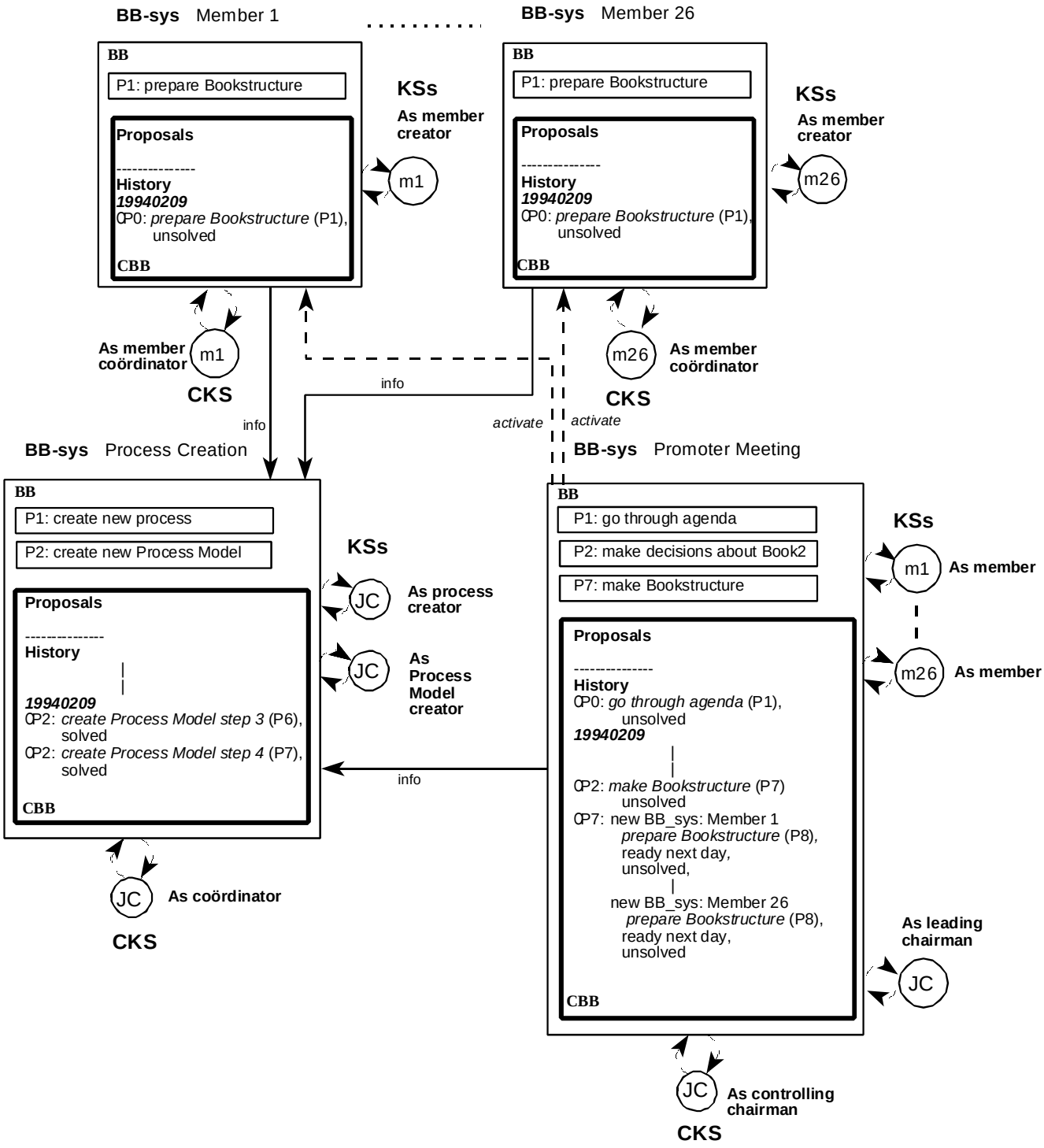


Fig 4.8 Process model step 4

In the BB-sys Promoter Meeting, the new problem *prepare Bookstructure* is created as a subproblem of *make Bookstructure*.
All members are asked to prepare a bookstructure separately. This implicates that every member needs a separate BB-system as every member, as creator and controller of this activity, has to be the only KS and the CKS involved in the solving of this new problem.

The process model (Fig 4.8) shows the new BB-systems. The new BB-systems are activated by the BB-system *Promoter Meeting* and they all have the unsolved 'initial' problem *prepare Bookstructure* on the BB.

Section (13) of the event trace of step 4 (Fig 4.9) presents the creation of all new BB-systems and the activation of all KSs and CKSs involved. In section (14), the *HistoryList* of *Promoter Meeting* is updated.
Section (15) of the event trace shows how JC as process model creator from the BB-system *Process Creation* asks information from the BB-systems *Promoter Meeting, Member1, ...* and *Member 26* before he creates the process model of step 4.

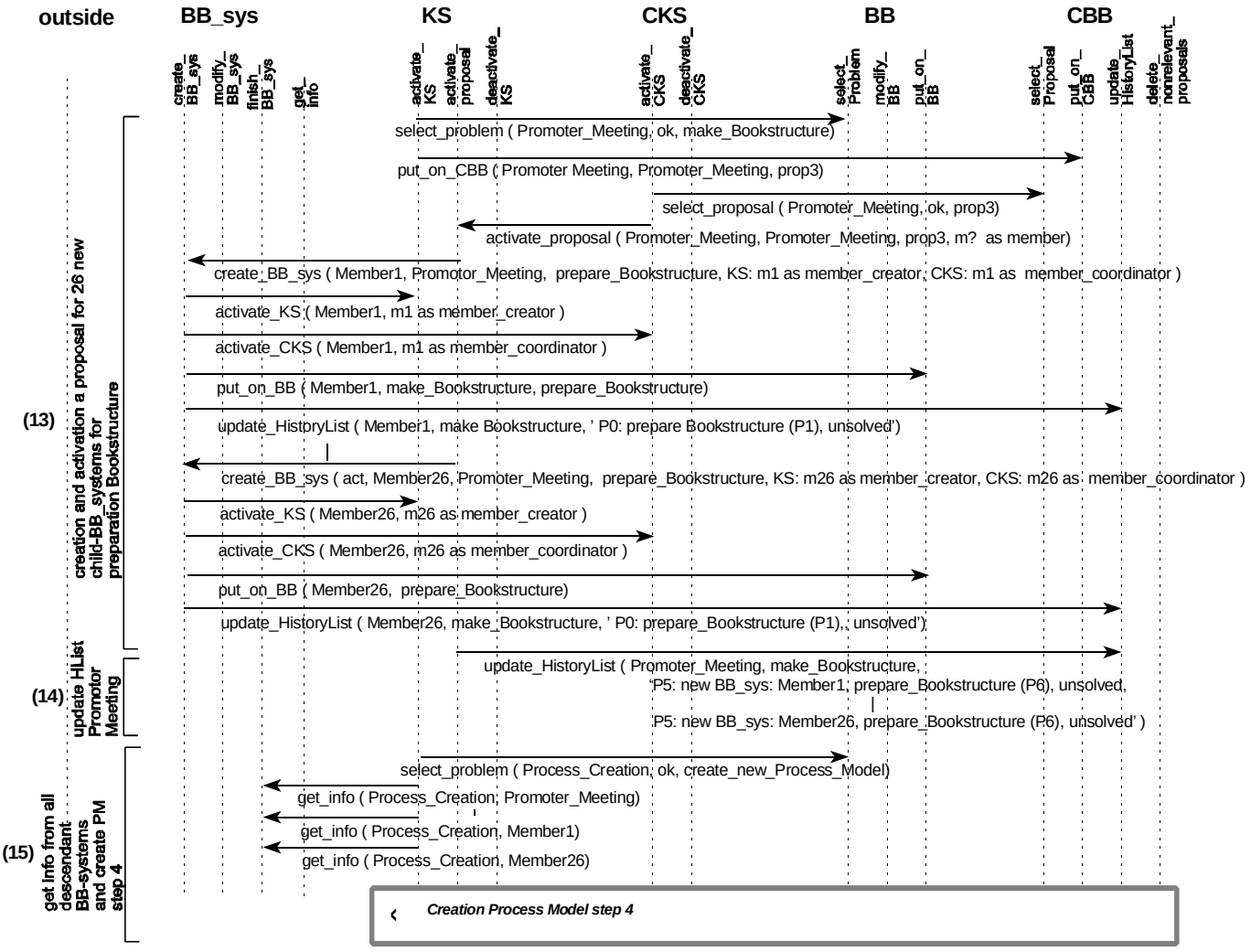


Fig 4.9 Event trace step 4

4.5.5 Step 5: BB-system Promoter Meeting receives the results of the child-BB-systems

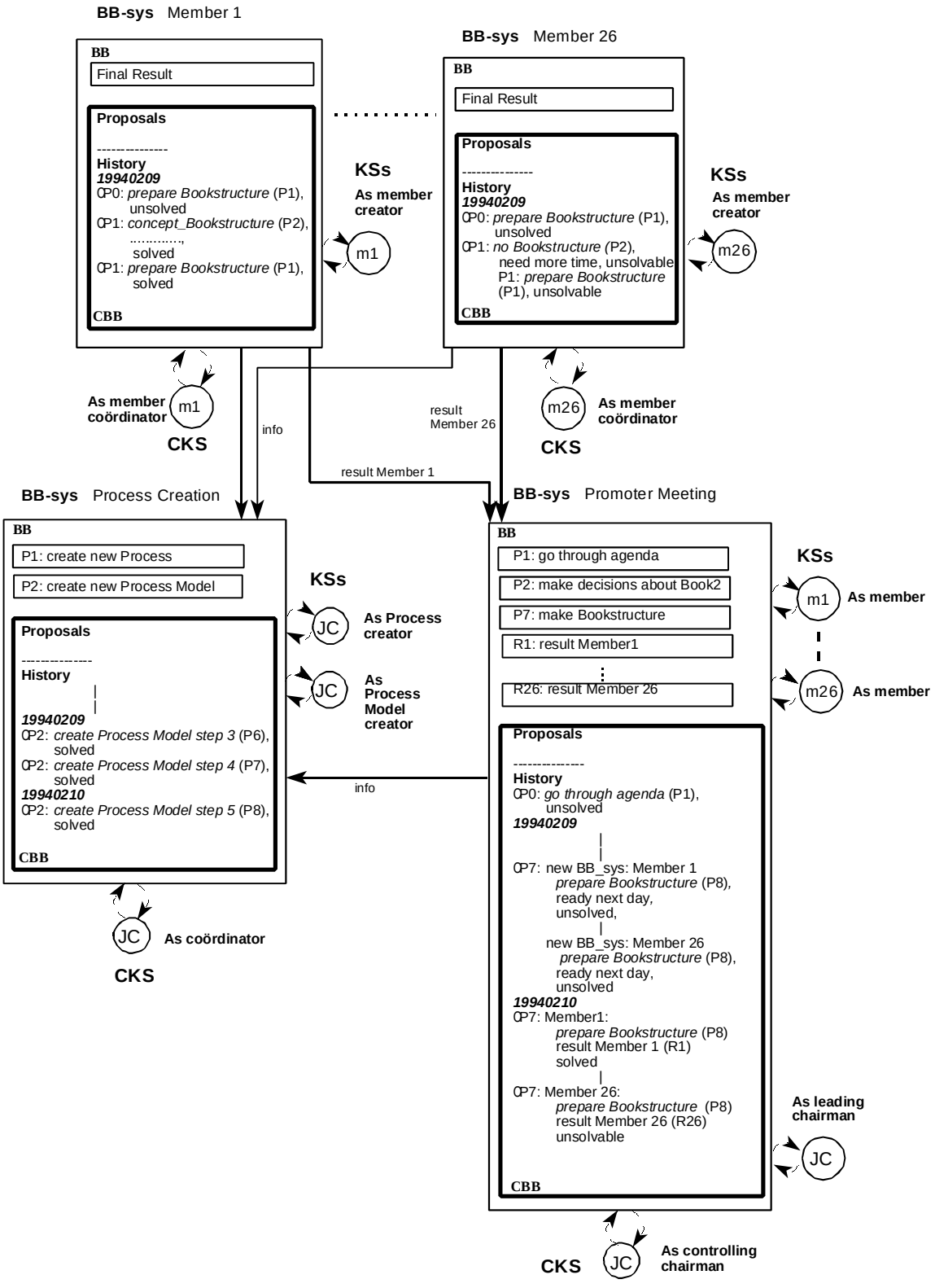


Fig 4.10. Process model step 5

The process model of step 5 (Fig 4.10) shows that member 1 has found a solution to the problem *prepare Bookstructure*. The BB-system of member 26 indicates that member 26 has declared the initial problem unsolvable. The reason is that the KS, member 26 'needed more time' to come to a solution.
There may also be members that still have the problem *prepare Bookstructure* in an unsolved state. The given example says that there are two proposals for a possible structure, so, on all BB-systems of the members together, there must be two solutions.
The CKS of every member puts the result of its BB-system as a proposal on the CBB of the BB-system *Promoter Meeting*. The CKS of *Promoter Meeting* puts the result of every child-BB-system on the BB.

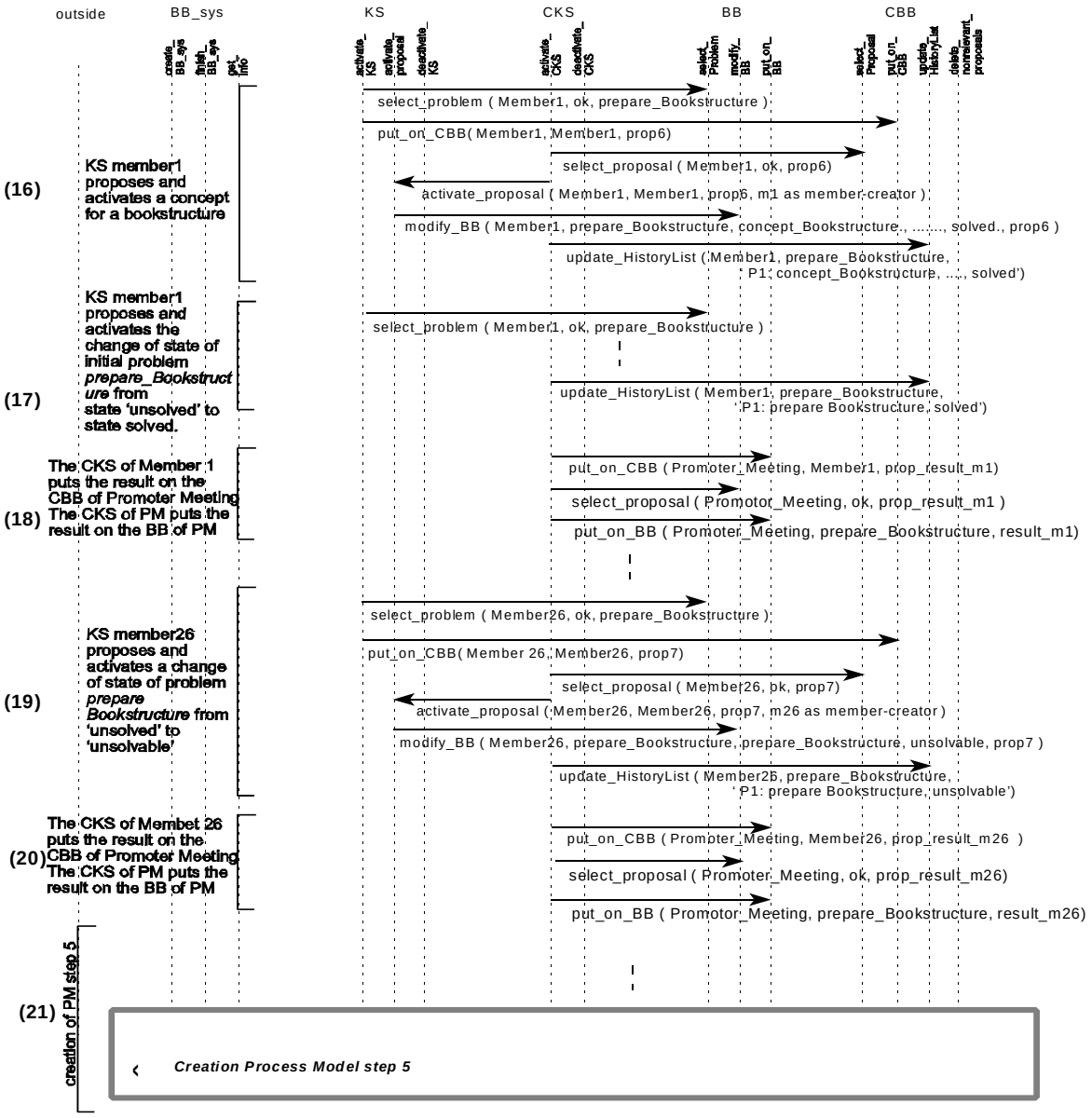


Fig 4.11 Event trace step 5

The event trace of step 5 is presented in Fig 4.11.
In the event trace, the activities on the child-BB-systems *Member1* are recorded in sections (16) and (17) and the activities of *Member 26* in section (19). In sections (18) and (20), the CKS of the parent-BB-system *Promoter Meeting* receives the input from *Member1* and *Member 26*.

Note that the KS of *Member 1* needs two sections, section (16) and section (17), to come to a result. In section (16), he creates a new 'solved' subproblem, *concept Bookstructure*. Later, he will come to the conclusion that this subproblem has also solved the 'initial' problem, *prepare Bookstructure*. In section 17, the state of *prepare Bookstructure* is changed from 'unsolved' to 'solved'.
Member 26 only needs 1 section , section (19), to come to a result, as *Member 26* executes two actions by only making one call for *BB_sys.modify_BB_sys*. First, he will add an 'unsolvable' subproblem to the BB, *no Bookstructure*, and later he also changes the state of 'initial' problem *prepare Bookstructure* from 'unsolved' to 'unsolvable'. As a modification of the BB can consist of more actions, this modification by the KS of Member 26 is permitted.
As Member 1 to Member 26 are intended to solve the same problem in parallel, the sequence of events as represented in fig 4.11 is not a very 'probable' sequence of events.

4.5.6 Step 6: Processing the results of child-BB-systems and the termination the child-BB-systems.

In step 6, a decision has to be made about the book structures, prepared by the members. Finally, the chapter structures have to be made. First, the chapter groups are formed. Later, every group starts working on the chapter structure of the chosen chapter.

The process model of step 6 is presented by Fig. 4.12 and the event trace of step 6 is presented in Fig. 4.13.
On the BB of *Promoter Meeting* are the results of the BB-systems *Member 1 - Member 26*. A KS proposes and activates the decision that the result of *Member 1* , R1, is the best result and with a small modification, this book structure is accepted as the final book structure.
The original problem *make Bookstructure* is declared 'solved'.
As *make Bookstructure* is solved, all the results of the other BB-systems can be deleted.
Finally, the BB-systems *Member 1 - Member 26* are to be terminated. In section (25) of the event trace, the finishing of *Member 1* is worked out.

The event trace of step 6 only represents parts of the complete event trace as most actions are already worked out in the previous event traces.
The part of the termination of *Member 1* is worked out, as this is the only action of a BB-system that is not worked out before.
The sections (22), (23), (24), (26) and (27) all concern modifications of the BB. The section (28) represents the creation of a new BB-system. Section (4) of the event trace of step 2 already shows the creation of a new BB-system.

The second part of the event trace concerns the new subproblem *make Chapterstructure*. In order to work in small groups on the problem, a new subproblem *form groups* of problem *make Chapterstructure* is created.
When *form groups* is solved, a new BB-system, named *Chapter 9 Group* is created, so that the members Luuk, Vicenzo and Jacques can work separately on the chapter structure of chapter 9. During the formation of groups, it was already decided that Luuk should be the editor and Vicenzo and Jacques should be the authors of chapter 9. As Jacques already left the meeting, Jacques is not included as KS in the *Chapter 9 Group*.

Note that the BB-system *Chapter 9 Group* has two initial problems on the BB, *prepare structure chapter 9* and *prepare 1 or 2 sheets*.

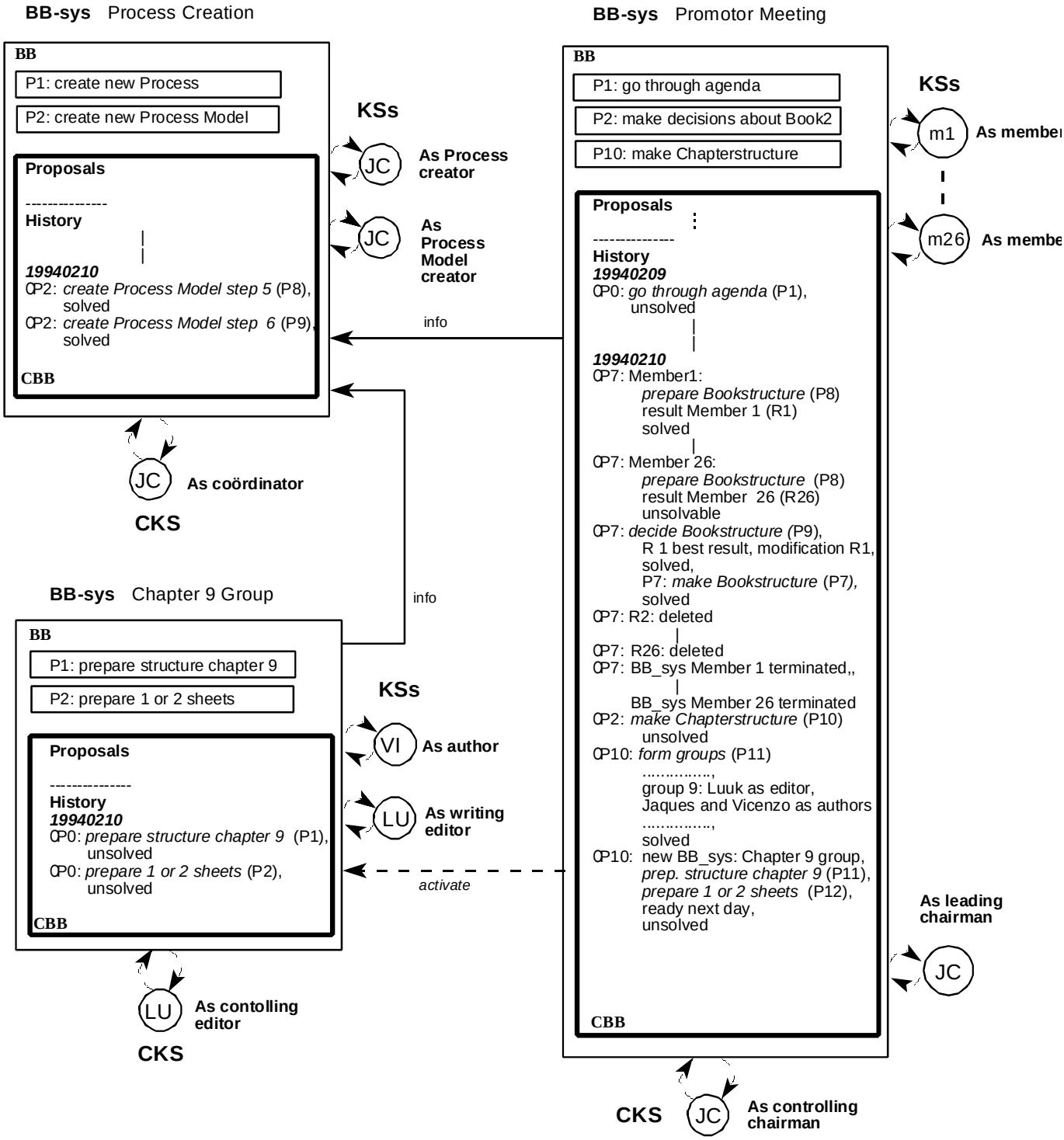


Fig 4.12 process model step 6

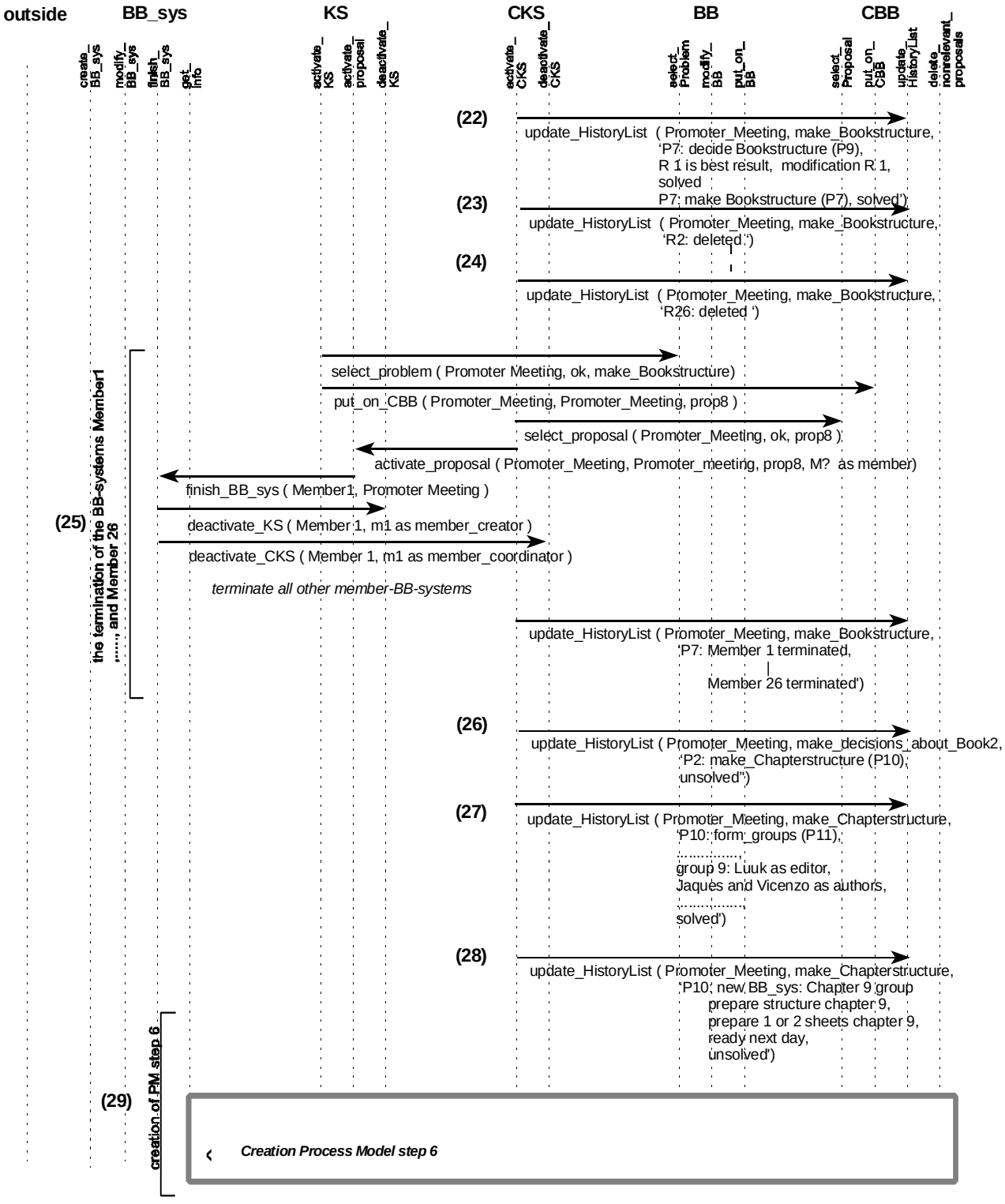


Fig 4.13 Event trace step 6

4.5.7 Step 7: A discussion on *Chapter 9 Group* and *Promoter Meeting* proposes its own termination.

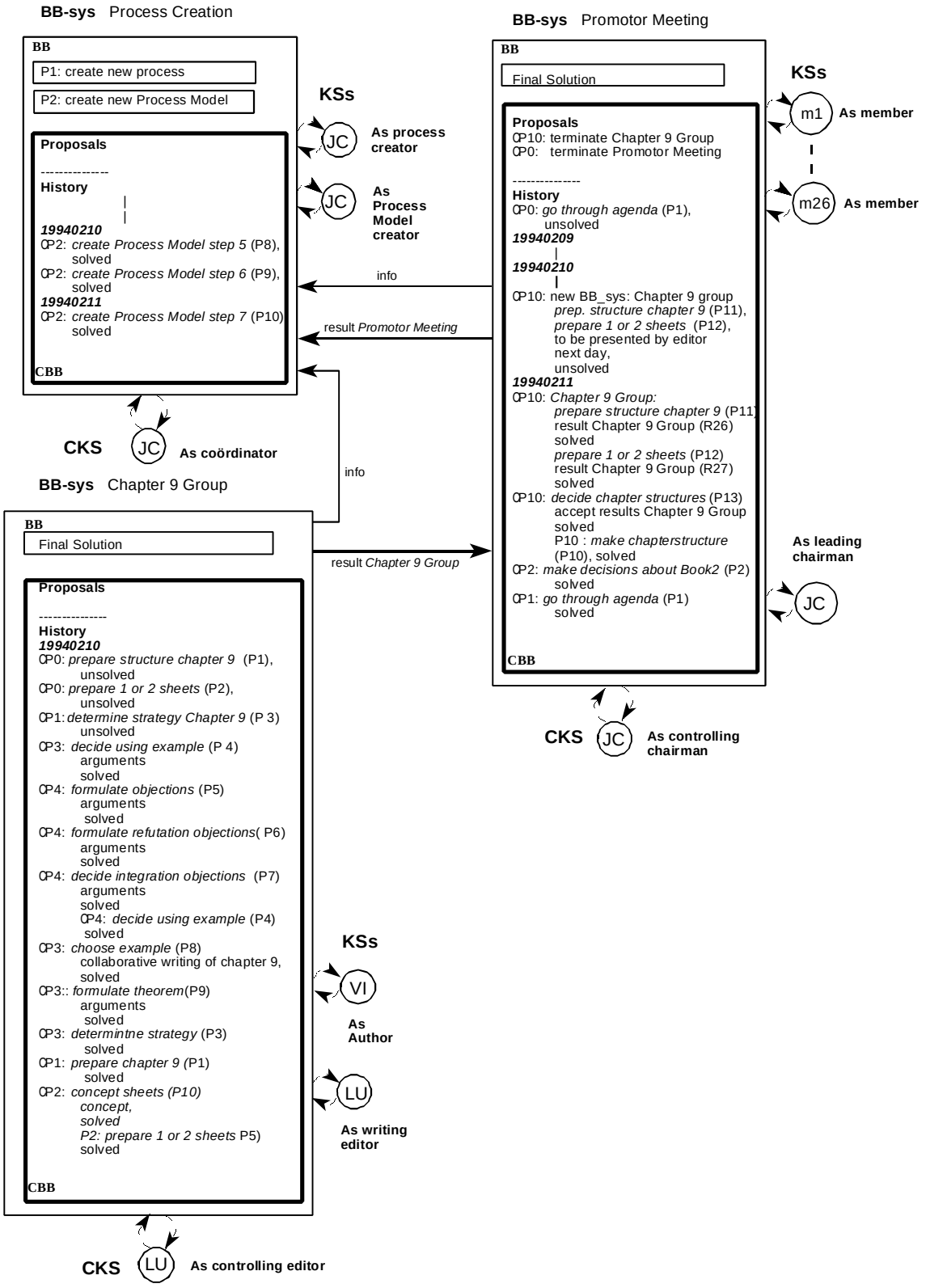


Fig 4.14 process model step 7

The given example relates the discussion of the group that prepares a chapter structure of chapter 9. The next day, the results of this discussion are related to the members of Promoter. The Promoter meeting agrees with the chapter structure of chapter 9.
By finishing the decision of the chapter structures, the Promoter meeting has made all necessary decisions to start up the second book. When all topics on the agenda are finished, the Promoter meeting is to be terminated.

All details of the given example of step 7 concern features of the SOCCA model that are already worked out in former steps.
The event trace of step 7 is therefore omitted.

The process model of step 7 (Fig. 4.14), shows all details processed during step 7.
At the moment the process model is drawn, 2 proposals are on the CBB of *Promoter Meeting*.
The first, is a proposal for the termination of Chapter 9 Group and the second is a proposal for the termination of Promoter Meeting. In step 8, these proposals are activated.

4.5.8 Step 8: BB-system *Chapter 9* changes its own 'initial' problem.

This step concerns all remaining details of the given example: the writing of chapter 9 of the second book

Step 8 starts with the actual termination of *Chapter 9 Group* and *Promoter Meeting*. Both proposals for termination were created by KSs of *Promoter Meeting*. Although both proposals concern the termination of a BB-system, they are handled differently. The termination of Chapter 9 Group is activated and executed by *Promoter Meeting*. This action is handled in exactly the same way as the termination of *Member 1* as described in section (25) of the event trace of step 6.
The termination of *Promoter Meeting* however, has to be activated by the parent, *Process Creation*. In section (30) of the event trace of step 8 (Fig. 4.16), the termination of *Promoter Meeting* is worked out.

Based on the results of *Promoter Meeting*, Knowledge Source JC as 'Process creator' of *Process Creation* creates a new process: the actual making of the second book. JC will create a new BB-system, called *Book 2* that is responsible for the writing of the second book. The members of the Promoter community will be involved in this new BB-system as members.
All decisions made by the members of *Promoter Meeting* with respect to the writing of the second book will now be part of the problem description of the 'initial' problem of Book 2, named, *make Book2*.
The first subproblem of *make Book 2* is *write round 1*.
Book 2 will then create a new separate BB-system *Chapter 9* that is responsible for the writing of chapter 9. *Chapter 9*'s initial problem is *write chapter 9*.
Nearly all details described in the given example concerning the writing of chapter 9, can be modelled as modifications of the BB of *Chapter 9*. These modifications can be traced back in the History of *Chapter 9* in the process model of step 8, Fig. 4.15.
The only exception is described in the last paragraph of the example: the changed set-up of chapter 9.

In terms of the SOCCA model, the changing of the set-up of chapter 9 is the changing of the problem description of the initial problem of *Chapter 9*.
As a modification of an initial problem is a modification of the BB-system involved, this modification can only be activated by the parent of the BB-system.
In section (31) of the event trace of step 8, this particular modification of the BB-system *Chapter 9* is worked out.

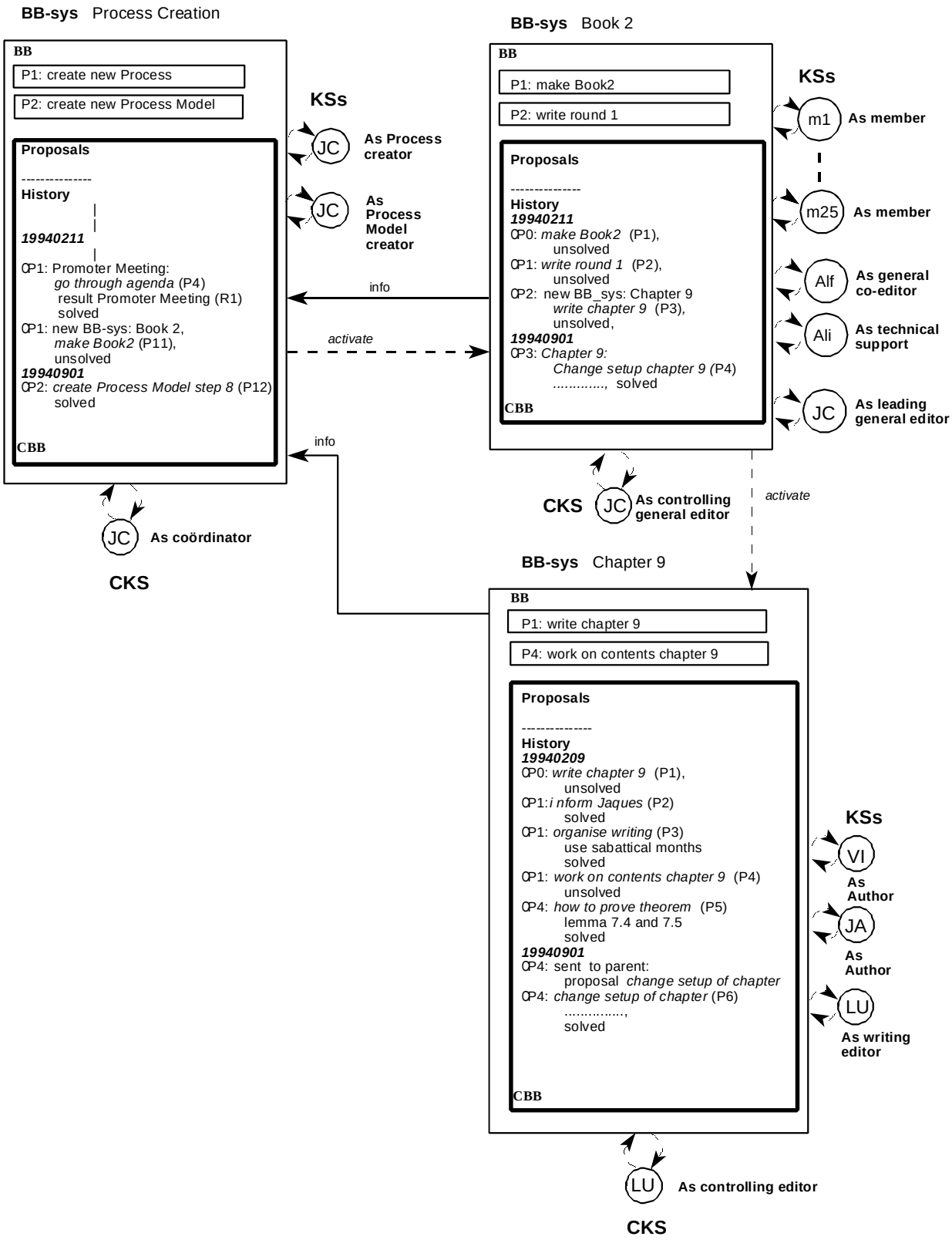


Fig 4.15 process model step 8

.

The event trace of section (31) starts when one of the KSs of *Chapter 9* has already created and posted a proposal for the modification of the initial problem. The CKS of *Chapter 9,* Luuk as controlling editor, selects this proposal and puts the proposal on the CBB of the parent-BB-system, *Book2*.
The CKS of *Book 2* , Jean Claude as controlling editor, selects this proposal and activates the proposed modification of *Chapter 9*.
The selection and activation of the proposal triggers 3 calls for *CBB.update_HistoryList*: the first call registers the transportation of the proposal to the CBB of the parent. By the remaining calls, the change of *Chapter 9* is registered by *Chapter 9* and *Book 2*. This way the Members of Book 2 are also informed of the changes of the set up of chapter 9.

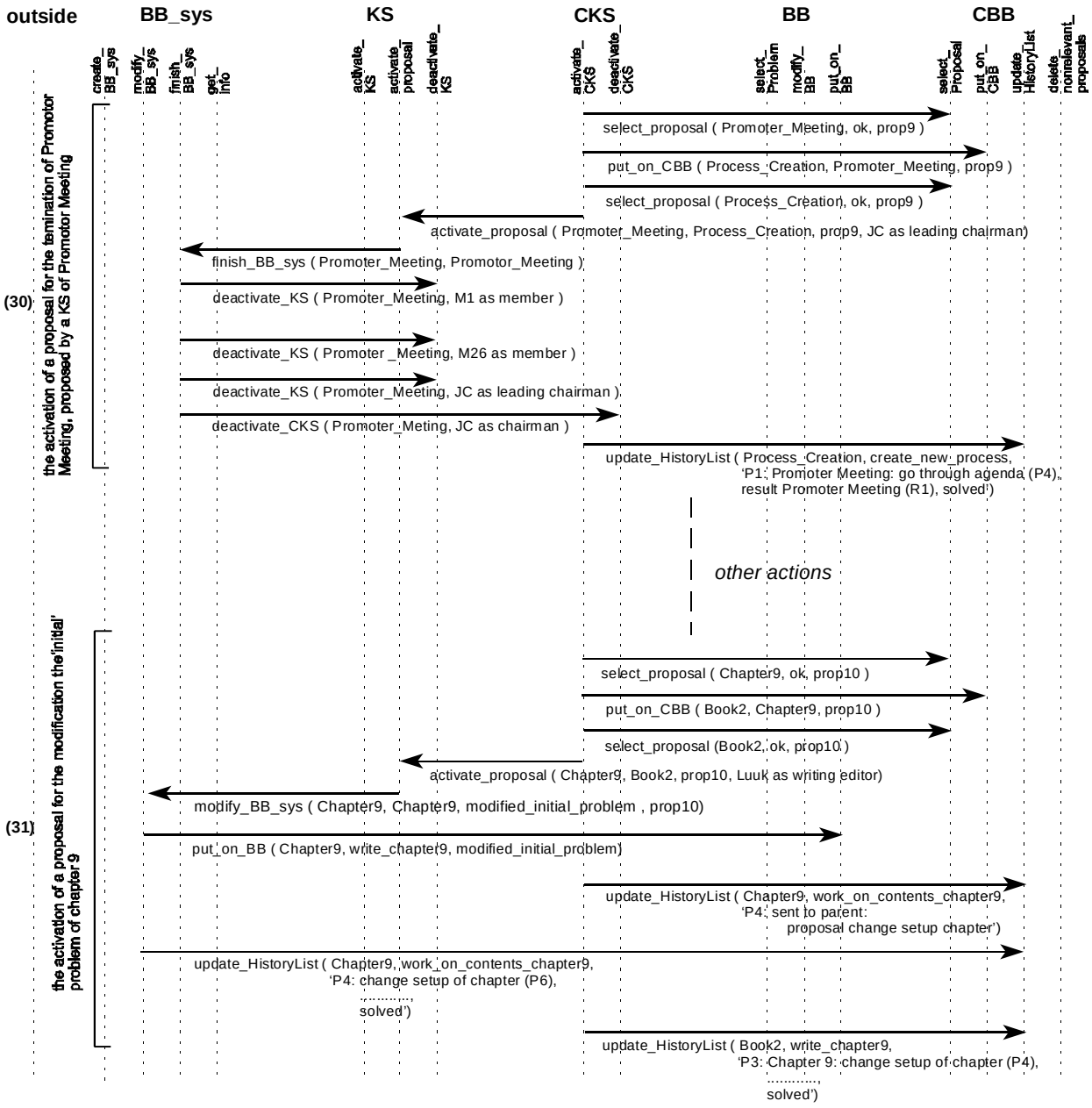


Fig 4.16 event trace step 8

4.5.9 Step 9: Parent *Book 2* formulates a second problem for BB-system *Chapter 9*.

This final step is not explicitly described in the given example. It is worked out to demonstrate an interesting feature of the Blackboard model that is not yet described in one of the previous steps.

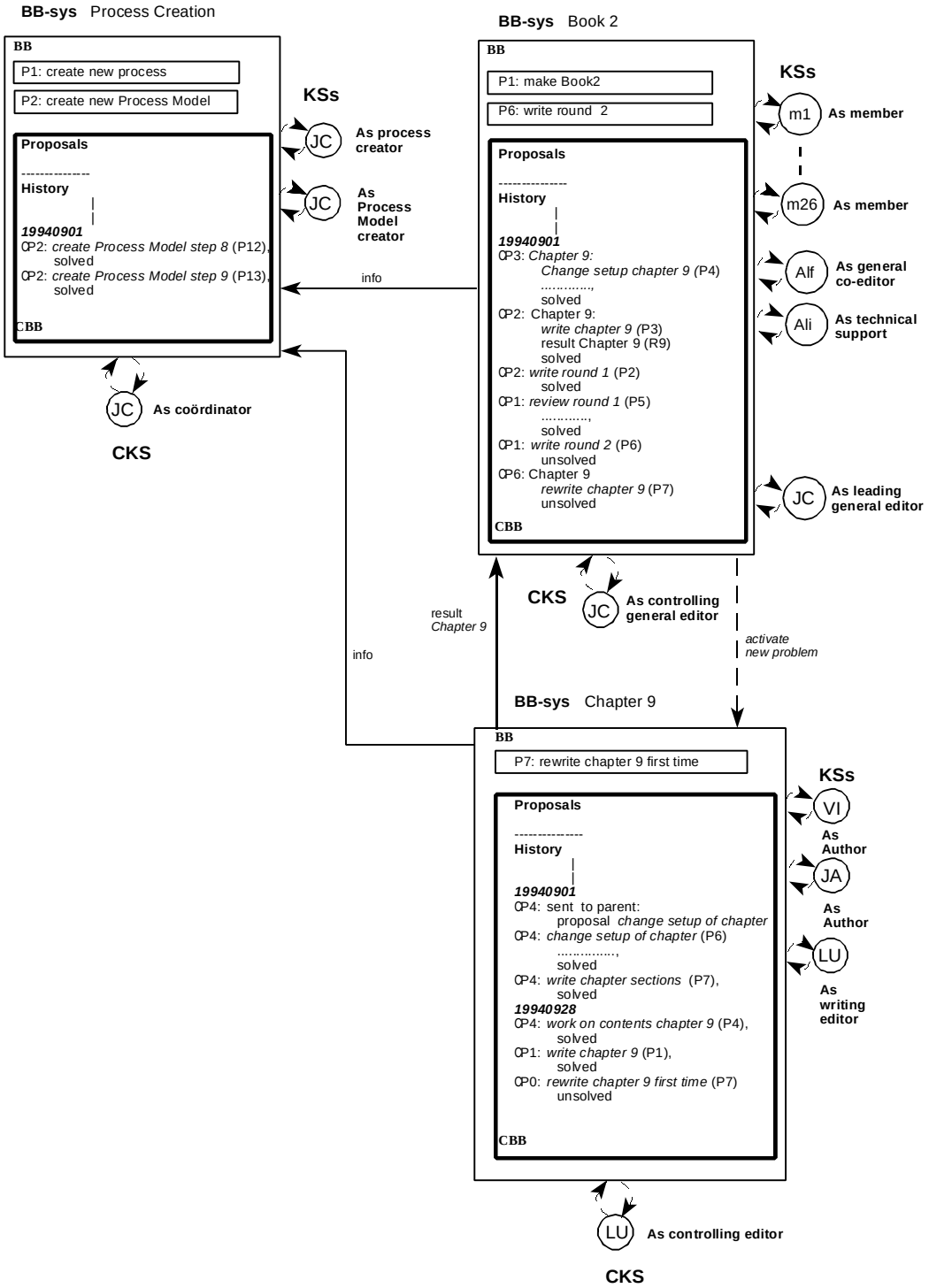


Fig 4.17 process model step 9

Chapter 9 has now solved the initial problem *write chapter 9* and put the result on the CBB of the parent, *Book2*.
The result is accepted and put on the BB of *Book 2*. Based on the result of *Chapter 9*, the subproblem *write round 1* is now solved. *Book 2* now starts up the review of chapter 9 by creating a new subproblem, named *review round 1*. As the given example relates no details of the review of chapter 9, this subproblem of *make Book2* is not worked out any further.
When *review round 1* is solved, a new subproblem of *write Book2*, named *write round 2* is started. *write round 2* has a new subproblem *rewrite chapter 9 first time* which is to be the new 'initial' problem of *Chapter 9*.
In this way, the result of *Chapter 9* is modified by *Book 2* and *Chapter 9* in turn.
In section (32) of the event trace of step 9, *Book 2* puts the new 'initial' problem on the BB of *Chapter 9*.

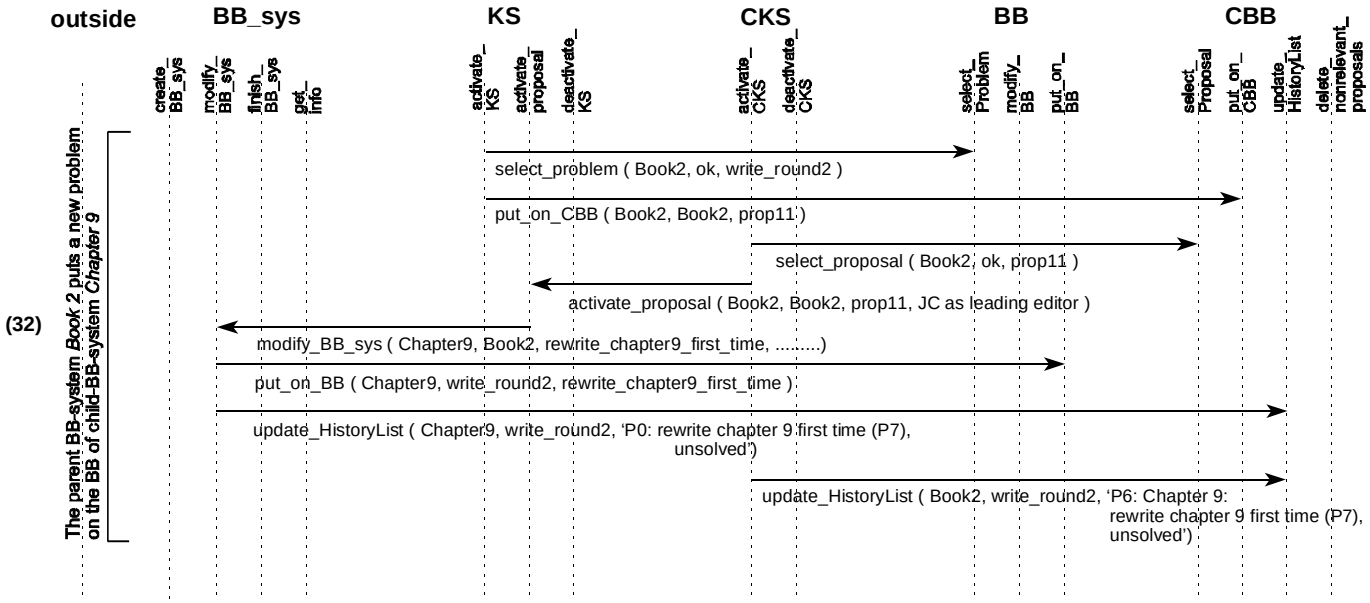


Fig 4.18 Event trace step 9

Part V: Conclusions and further research

Summarising, it can be concluded that non-automated Blackboard Systems can be modelled successfully in SOCCA.
By applying the Blackboard System to a 'real-life' example, it was demonstrated that the proposed Blackboard System is fit to model even complex human collaboration processes. The simple process models, that were created at fixed points in time, clearly visualise the evolution on the Blackboards by means of changes to the problems on the Blackboards and by means of changes to the Blackboard Systems involved. In the course of the complete process of the collaborative writing of the book, several different Blackboard Systems were generated of which some were terminated later.
In the event traces, the communication between the objects in their problem-solving activity is outlined. This way, the actual realisation of evolution is visualised.
Naturally, this evolution can also be visualised in other ways, depending on the type of information that is to be illustrated.

In the course of the design of the Blackboard-System, some interesting features of human behaviour were detected.
The notion of human roles is very important when humans are involved in processes. The refinement of human roles can sometimes clarify the human behaviour that is associated with the human role.
If the relevant, distinct roles of a human can be identified, even human behaviour can be simulated by means of a Blackboard System. The control of the Blackboard System will be represented by the human part that personifies the Control Knowledge Source. The coordination between the human roles is the responsibility of the Control Knowledge Source.

If human behaviour in relation to (evolving) human collaboration processes is to be modelled, this human behaviour or the coordination of the different human roles has to be modelled more explicitly. Also the use of multiple control of a Blackboard System needs to be studied further.

As the design of the Blackboard System emphasises its reproductiveness and the communication between the systems, the evolution on the Blackboard itself remains relatively underexposed. Especially when Blackboard Systems are used to investigate special cases of human collaboration processes, this evolution may need more refinemen

References

[1] Robert Engelmore and Tony Morgan, *Blackboard Systems*, Addison Wesley Publishing Company, 1988

[2] Promoter, *Software Process Modelling and Technology*, Anthony Finkelstein, Jeff Kramer and Bashar Nuseibeh (eds.), Research Studies Press LTD., Taunton, England, 1994

[3] Promoter, *Software Process: Principles, Methodology, Technology*, Jean Claude Derniame, A. B. Kaba and Brian Warboys (eds.), publisher and date of release not yet known

[4] Gregor Engels and Luuk Groenewegen, *SOCCA: Specifications of Coordinated and Cooperative Activities*, University of Leiden, Department of Computer Science, 1993

[5] Luuk Groenewegen, *Parallel Phenomena 1-14*, University of Leiden, Department of Computer Science, Technical reports, 86-20, 87-01, 87-05, 87-11, 87-18, 87-21, 87-29, 87-32, 88-15, 88-17, 88-18, 90-18, 91-19, 1986-1991

[6] Allen Newell, Some problems of the basic organization in problem-solving programs. In: *Proceedings of the Second Conference on Self-Organizing Systems*, Yovits, M.C., Jacobi, G.T., and Goldstein (eds.), pp 393-423, Spartan Books, 1962

[7] Daniel D. Corkill, National Science Foundation Phase I Final Report, *A Blackboard Based Collaboration Environment for Human Problem-Solving*, Blackboard Technology Group, Amherst, MA, 1996

[8] Tineke de Bunje, Gregor Engels, Luuk Groenewegen, Aart Matsinger, Mark Rijnbeek, *Industrial Maintenance Modelled in SOCCA*, Philips Research Laboratories and Leiden University, Computer Science Department,

[9] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser and D. Raj Reddy, *The Hearsay-II speech- understanding system: Integrating knowledge to resolve uncertainty*, Computing Surveys, 1980

Appendix A: Identification of BB-systems and problems of the given example

In this section, the BB-systems and their child-BB-systems and the problems and their subproblems of the given example are outlined.

Fig A.1 represents all BB-systems involved in the given example. The BB-system *Process Creation* is the root-BB-system. The BB-systems *Promotor Meeting* and *Book 2* are child-BB-systems of *Process Creation*. The BB-systems *Member1,...., Member 26* and *Chapter 9 group* are child-BB-systems of *Promotor Meeting*. Finally, *Chapter 9* is child-BB-system of *Book 2.*

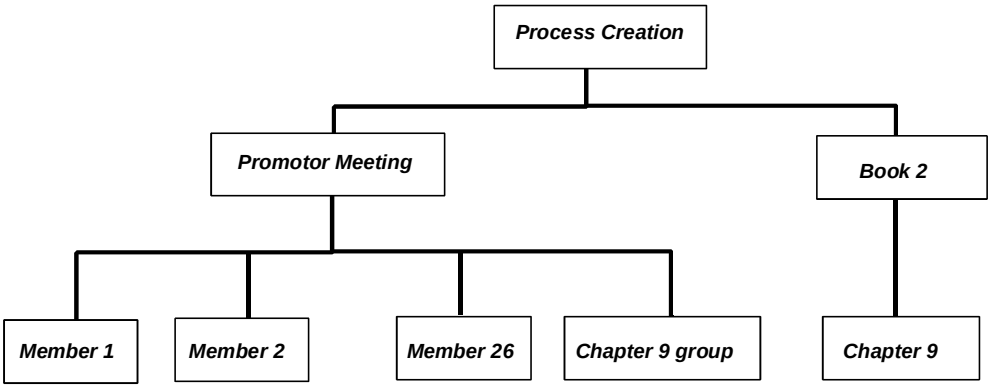


fig A.1. All BB-systems of the example

Every Blackboard system has its own tree of subproblems. The name of a Blackboard System is printed at the top of the tree. The root of the tree of subproblems is the 'initial' problem. If a BB-system has more than one 'initial' problem, every 'initial' problem has its own tree of subproblems.
If a subproblem causes a child-Blackboard System, the name of the child-BB-system is written below the subproblem.

In fig A.2, the two initial problems of the root-BB-system and their subprocesses are presented.
The initial problem create new Process creates two new subproblems or processes named 'go through agenda' and 'make Book2'.
'go through agenda' causes the new BB-system *Promotor Meeting* and the subproblem 'make Book 2' causes the child-BB-system *Book 2*.

Fig A.2. also presents the subproblems of *Promotor Meeting* and *Member 1, ..., Member 26*.
Finally fig A.3. presents the subproblems of *Chapter 9 group*, *Book2* and *Chapter 9*.

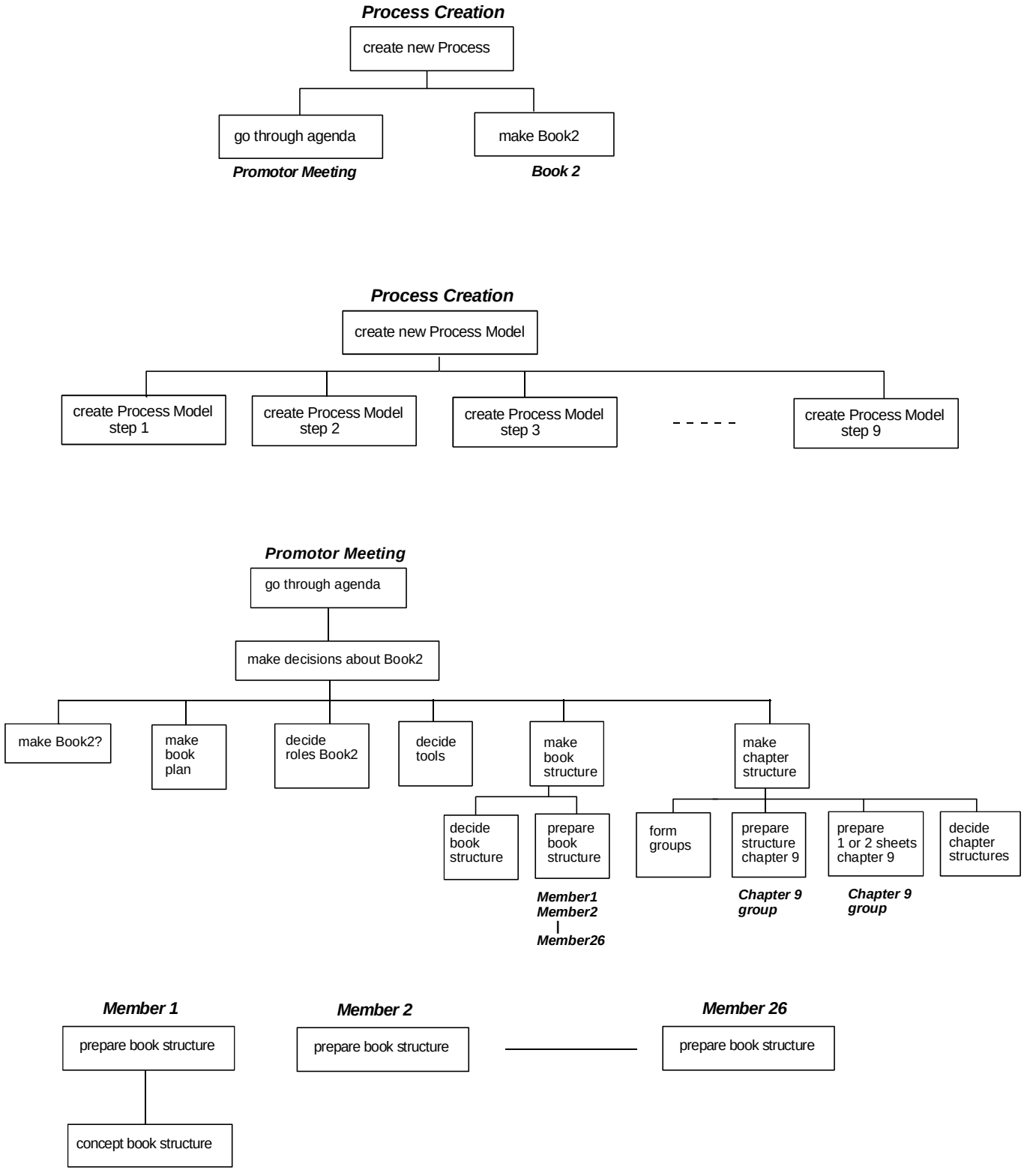
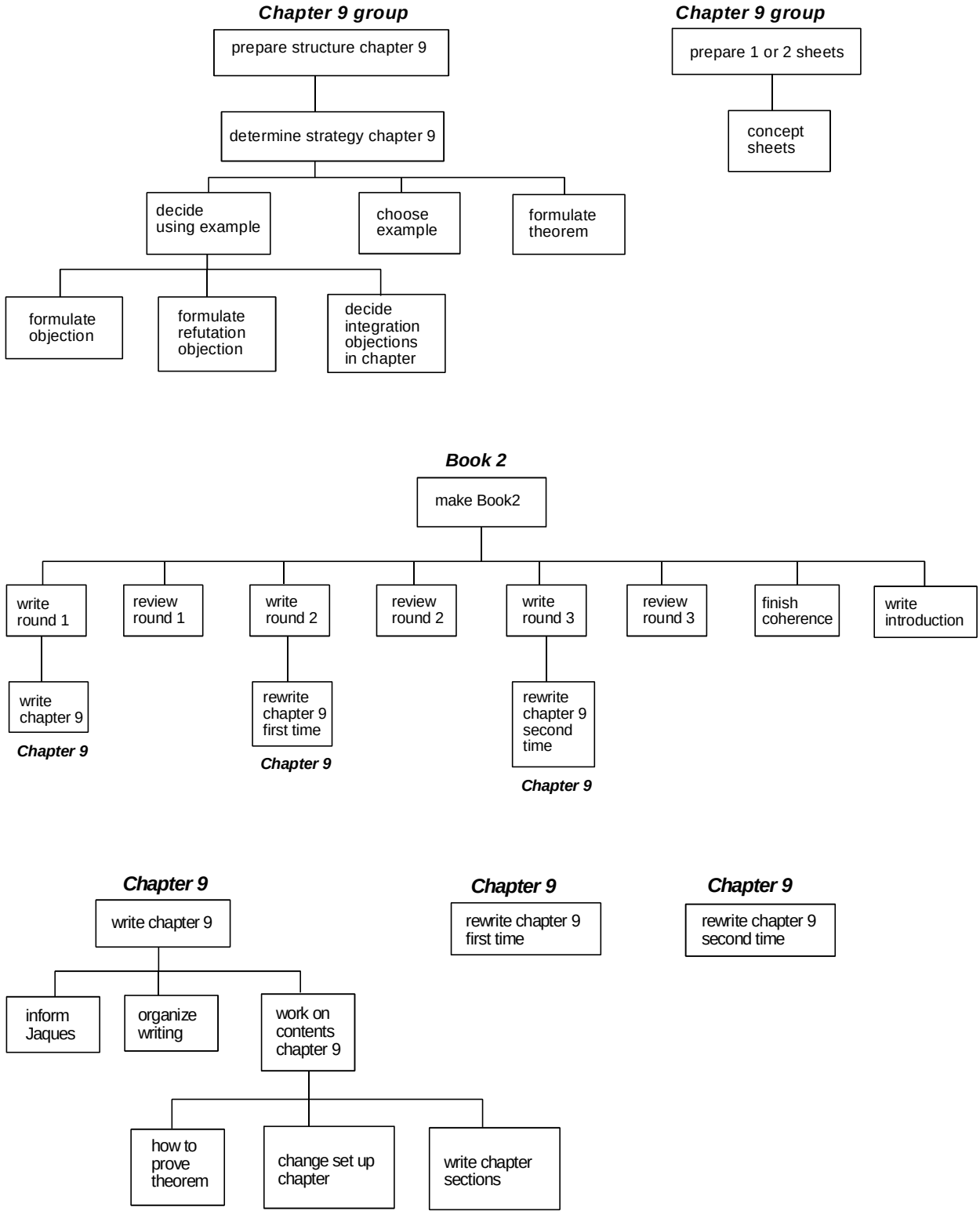


fig A.2. subproblems of the BB-systems *Process Creation, Promotor Meeting* and *Member1, ..., Member 26*



figA.3. Subproblems BB-systems *Chapter 9 group, Book 2*