The Criticality of Modeling Formalisms in Software Design Method Comparison *

Rodion M. Podorozhny and Leon J. Osterweil

August 16, 1996

email: **{podorozh|ljo}@cs.umass.edu**
Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

This paper describes experimentation aimed at making the comparison of software design methodologies (SDM's) more of an exact science. Our aim is to lay the foundations for this more exact science by establishing fixed methods and conceptual frameworks that are able to assure that comparison efforts will yield predictable, reproducible results. Earlier papers have proposed the use of a systematic process to compare SDM's. This process assumes that the comparison will be done relative to a fixed standard SDM feature classification schema, and with the use of a fixed formalism for modeling the SDM's. Early experiments with this approach have yielded interesting SDM comparisons, but have raised questions about how sensitive these results might be to the choice of modeling formalism. In this paper we study this sensitivity by varying the choice of modeling formalism. We describe an experiment in which we fix a pair of SDM's and then use two different formalisms to obtain two different comparisons of that pair of SDM's. We then compare the comparisons. Our results suggest that comparison results may be relatively insensitive to differences in modeling formalisms. This paper also suggests an approach to further experimentation.

Keywords Software Development Methodology, Software Process, Process Formalism, Comparison, Base Framework

*This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract F30602-94-C-0137.

1 Introduction

1.1 Background

In earlier work [6], [9] we have argued that processes should be viewed as artifacts that can be thought of as emerging from a process-development activity. As developed products, it seems only natural to consider what might be necessary in order to assure that these products are developed effectively, and that they are of demonstrably high quality. In short, the realization that processes are products suggests the importance of a discipline of engineering them.

While this may seem to be an attractive notion, it is complicated by the fact that processes are products that differ from conventional manufactured products in some significant and daunting ways. Processes are not tangible. They cannot be seen, touched, or accessed by any of the five human senses. In that they have no physical manifestations, they obey no laws of mechanics, and cannot be measured in conventional ways. While certain dynamic properties and behaviors may be attributable to them, they should not be expected to obey Newtonian Laws of Motion.

Instead, processes must be thought of as being products of thought processes. They are intellectual products. They can be sensed, understood, and evaluated only by indirect means through their effects and manifestations. While these characteristics differentiate processes from most conventional products, it seems to us that they also place processes in a category with computer software another type of product that we know well, and for which we have devised important engineering tools. Like processes, computer software is invisible and intangible. It has important dynamic properties, yet obeys no laws of Physics and defies physical measurement.

In earlier work we have suggested that many of the ideas, approaches, techniques, and formalisms of software engineering should be applied to the engineering of processes as well. Our earlier work has evaluated this suggestion by applying a range of software engineering techniques to process engineering. In [6] we suggested that software development environments should be viewed as systems for supporting the development, execution, evaluation, and evolution of software development processes. In [11] we suggested that programming languages should be used to program the process of developing software, and presented evidence that a specific language we had developed showed promise of being effective in supporting this. In [10], [7] we suggested that software design notations and software process programming formalisms could be useful in establishing baselines that could be effective bases for the objective classification and comparison of processes. This paper builds upon that latter work and provides further evidence of the plausibility of this suggestion.

1.2 Discussion of Problem

We believe that one of the hallmarks of a mature scientific or engineering discipline is its ability to support the comparison and evaluation of the artifacts with which it deals. Comparison, in turn, seems to rest upon classification. Thus we believe that the establishment of a discipline of process engineering requires the development of techniques and structures for supporting the classification,

comparison, and evaluation of processes. In our earlier work we have proposed such techniques and structures, and have demonstrated their use by carrying out classifications and comparisons of processes drawn from the narrow and specialized domain of software design processes.

In [10] we have proposed CDM (Comparison of Design Methodologies) and presented a model of this process for the comparison of design processes. In that work we have also proposed the use of BF, a classification schema for organizing the key components of design processes. In [9], [10] we demonstrated the use of CDM and BF in comparing some software design processes. In [7] we have suggested that CDM and BF must be considered to be only initial suggestions, and that considerable community participation in evaluating and migrating them is necessary, based upon an ongoing series of design process classification and comparison activities. In particular, it seemed clear that others might suggest classification schemas other than BF. We also noted that the results we had obtained were based largely upon the use of a single formalism for modeling the design processes that we compared. The use of that modeling formalism, HFSP [5], seemed to enable us to obtain a range of interesting, useful, and credible design process classification and comparison findings. In the spirit of our earlier paper [7], however, we did feel that it was important to evaluate the use of HFSP as we were concerned that the comparison results obtained might be biased by the use of HFSP as the basis for these comparisons.

Thus, we embarked upon a new set of experiments aimed at attempting to evaluate the sensitivity of the classification and comparison results we obtained to the choice of modeling formalism selected. Specifically we repeated some of the design process comparison experiments (e.g., comparing Booch Object Oriented Design (BOOD) [2] to Jackson System Development (JSD) [3]) using CDM and a single fixed classification schema, but substituting a different modeling formalism for HFSP. In these new experiments we used SLANG [1] to model the design processes. We then compared the comparison results obtained using SLANG to comparison results obtained using HFSP. In this paper we describe the results obtained. We also offer this paper as evidence of the feasibility of systematically evolving a maturing discipline of software process engineering.

2 Approach

2.1 Comparison of the results of SDM comparison as a way to determine the influence of the modeling formalism used

Fig. 1 models an SDM comparison process that is essentially the same as CDM, described in [10]. But this model emphasizes the major functional components in the process. This functional decomposition provides a conceptual framework that seems convenient as the basis for establishing a discipline of SDM comparison. The figure also shows dependencies between these functional components. Note that although the diagram is strictly sequential, the process of comparison may be iterated using different formalisms and classification schemas. The figure highlights the critical role played by the choice of MF, the process modeling formalism and CS, the feature classification schema. Any formalism allows us to see only those components of an SDM that can be expressed in it. A classification schema allows us to compare only the components in those classes and categories

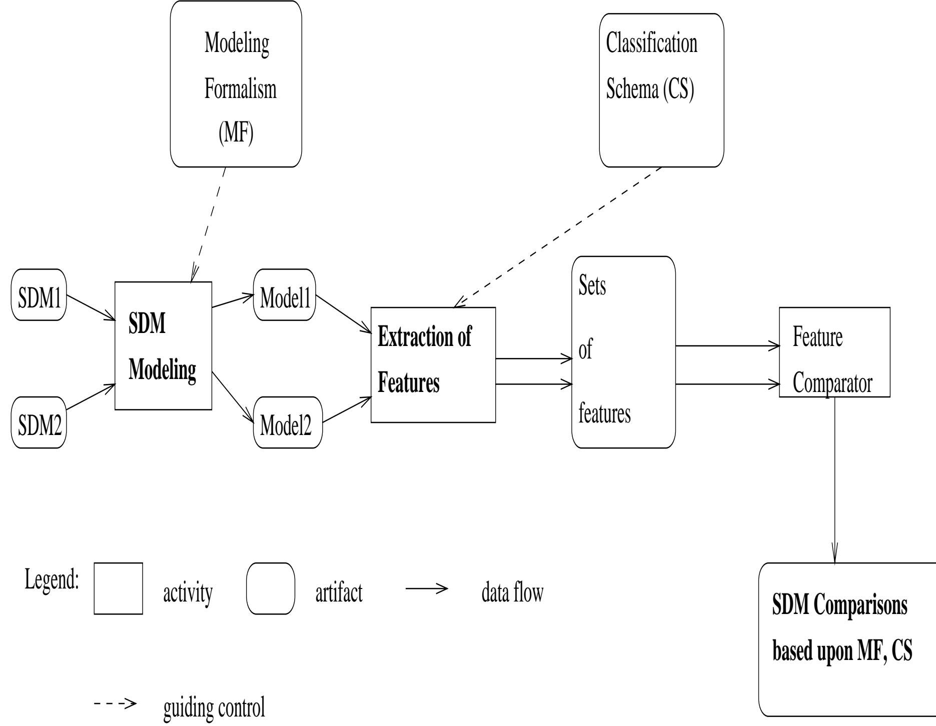

Figure 1: Model of CDM applied to two SDMs

that it includes. Thus, if an SDM has an aspect not captured by the formalism and/or schema, that aspect will not be considered and hence comparison results may be skewed and/or inaccurate.

We now use functional notation to express the comparison process of Fig. 1 more rigorously. The process consists of three principal functional transformations:

- $\boldsymbol{SDM_Modeling_{MF}} : \boldsymbol{SDM} \rightarrow \boldsymbol{SDM_Model_{MF}}$, where SDM is the space of all SDMs and SDM_Model_{MF} is the space of models of SDMs in the modeling formalism MF.
- $\boldsymbol{Extract_Features_{MF,CS}} : \boldsymbol{SDM_Model_{MF}} \rightarrow \boldsymbol{Feature_Structure_{CS,MF}}$, where $Feature_Structure_{CS,MF}$ is the space of all feature sets, structured by CS.
- $\boldsymbol{Feature_Comparator_{CS}} : \boldsymbol{Feature_Structure_{CS,MF}} \times \boldsymbol{Feature_Structure_{CS,MF}} \rightarrow \boldsymbol{SDM_Comparisons_{CS}}$, where $SDM_Comparisons_{CS}$ is the space of comparisons of features identified by CS.

We can now define the entire process of using CDM, MF, and CS to compare two SDMs as:

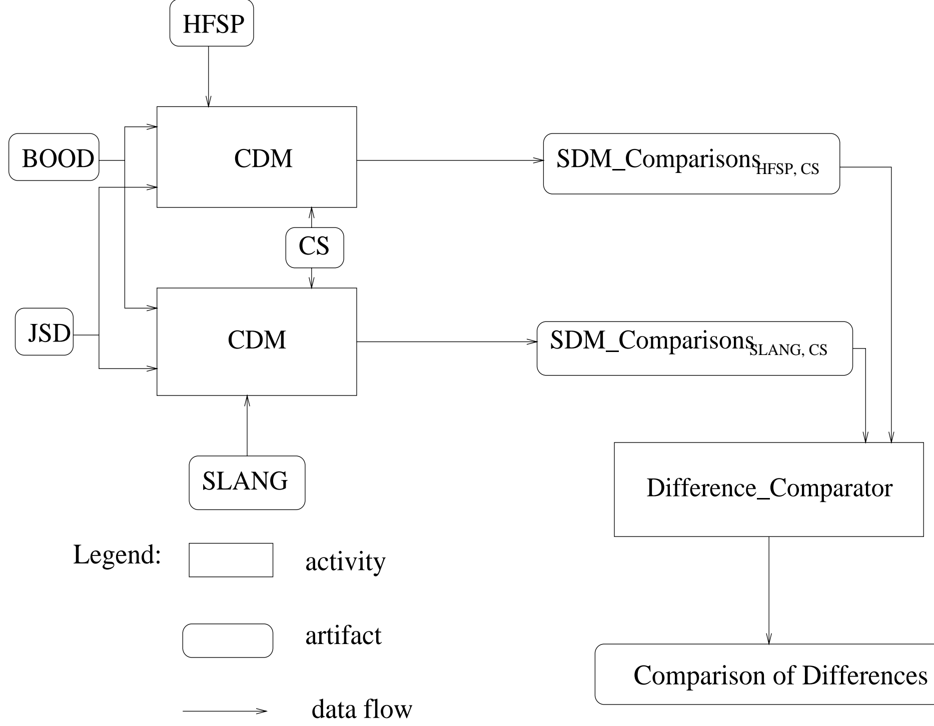

Figure 2: Model of our Experiment

$\boldsymbol{CDM_{MF,CS} : SDM \times SDM \rightarrow SDM_Comparisons_{CS}}$,

The $\boldsymbol{SDM_Modeling}$ function of CDM entails construction of models of SDMs. In the $\boldsymbol{Extract_Features}$ phase the features of the SDMs are extracted from the SDM models, based on the classification schema used. The last phase, $\boldsymbol{Feature_Comparator}$, takes two structures of features, one per SDM model, compares them, and outputs a comparison of the two feature structures. In this research we attempted to study the sensitivity of $\boldsymbol{CDM_{MF,CS}}$ to differences in the MF. Thus, let us represent the process of comparing JSD and BOOD based on a specific chosen MF and CS as the function:

$\boldsymbol{CDM_{MF,CS}(JSD, BOOD)}$.

Then, let us define a new function:

- $\boldsymbol{Compare_Differences_{CS} : SDM_Comparisons_{CS} \times SDM_Comparisons_{CS} \rightarrow Comparison_Differences_{CS}}$,

where $Comparison_Differences_{CS}$ is the space of distances between pairs of comparisons based

on CS, as measured by some distance measure. Then this research is aimed at computing

- $\boldsymbol{Compare_Differences_{CS}(CDM_{HFSP,CS}(JSD, BOOD),}$ $\boldsymbol{CDM_{SLANG,CS}(JSD, BOOD))}$

for some function measuring difference between CS-based comparisons. Fig. 2 is a model of this activity. In other words, we are trying to begin to assess how much the choice of the modeling formalism affects the results of software methodology comparison. This formula is currently conceptual, since we do not provide any rigorously defined numerical measurement of the differences between comparisons at this time. Though we do not have such a rigorous measure, we nevertheless can and do subjectively assess the difference between comparisons informally and anecdotally in this paper.

2.2 The Classification Schema used

Ideally, a comprehensive classification schema should include all the details of all the features of a complete universal SDM. In that case, we would be sure that no comparison could overlook any features of the SDM models being compared. While it is doubtful that such an ideal classification schema can be created, we believe it is feasible to develop a sequence of approximations to this ideal through iterative creation and modification. Each time a new or different SDM feature is encountered, we envisage classifying it and putting it in the proper place in the classification schema. Thus, analysts performing comparisons should be prepared to check whether all the features of compared SDM models are captured by the classification schema used and suggest modifications needed. Ideally, this would be done as a community activity leading to an increasingly broadly accepted classification schema.

Indeed, it is clear that this activity is already underway. Thus, for example, in earlier work [8] we have proposed BF. More recently Brinkkemper et al. [4] have suggested a different classification schema which they refer to as a supermethod. A careful comparison of BF and Brinkkemper et al.'s supermethod is beyond the scope of this paper, but it seems clear that the supermethod is more detailed, but less structured than BF. To underscore the generality of the Compare_Differences analysis suggested here, we adopt the features in Brinkkemper et al.'s supermethod as the basis for our comparison of SLANG-based and HFSP-based comparisons of BOOD and JSD. We will denote Brinkkemper et al.'s supermethod as BSM. BSM encompasses an extensive set of SDM features. Brinkkemper et al. call them concepts and divide them into four classes: static modeling, dynamic modeling, functional modeling, and implementation modeling. For the sake of consistency with our earlier work, we will continue to use the term "features" rather than "concepts" in this paper.

2.3 The formalisms used

As mentioned above, our experiment entails measuring the sensitivity of the results of the SDM comparison to the choice of the modeling formalism used. In order to experimentally study this we

sought two MFs having differences that seemed relatively small, but still significant. We selected HFSP and SLANG because both rest upon the centrality of the major concept of activity. Both support hierarchical activity decomposition. Both support specification of artifact flows through activities. On the other hand HFSP supports artifact decomposition, while SLANG does not. SLANG supports specification of timing constraints while HFSP does not. HFSP is textual, while SLANG is graphical. Thus we believed that HFSP and SLANG met our requirements for a pair of MFs that are close to each other, while retaining some significant differences.

3 Analysis of comparisons

This section contains the comparisons of JSD and BOOD for some selected features of BSM. We show how SLANG and HFSP were used to identify SDM components relevant to the selected BSM features, and how the comparisons were then performed. We then analyze the differences between the two comparisons.

3.1 Selection of BSM features to be compared

The number of features in BSM is quite large and space does not permit us to present comparisons for all of them here. Therefore, we present a selected subset of BSM chosen to typify the range of expected differences in comparisons.

3.2 Comparison of JSD and BOOD comparisons

The comparison presentations follow the process outlined in Fig. 1. The first step of the process (SDM_Modeling) implies the execution of the functions: [1]

$\boldsymbol{SDM_Modeling_{HFSP}}$ (JSD), yielding JSD_Model_{HFSP}

$\boldsymbol{SDM_Modeling_{SLANG}}$ (JSD), yielding JSD_Model_{SLANG}

$\boldsymbol{SDM_Modeling_{HFSP}}$ $(BOOD)$, yielding $BOOD_Model_{HFSP}$ and

$\boldsymbol{SDM_Modeling_{SLANG}}$ $(BOOD)$, yielding $BOOD_Model_{SLANG}$

JSD_Model_{HFSP} and $BOOD_Model_{HFSP}$ were presented in [8] and are reproduced here as Appendices A.1 and A.2. JSD_Model_{SLANG} is presented in Figs. 3– 7 and $BOOD_Model_{SLANG}$ is presented in Figs. 8– 12.

The second step of the process in Fig. 1 entails feature extraction, the third entails feature comparison. We now summarize our work in carrying out these two steps for each of the three selected

[1] It should be mentioned that the descriptions of the JSD and BOOD methodologies used in the comparison are now outdated, but for the purposes of this experiment it was essential that all models be constructed from the same descriptions. The descriptions used were the ones in [10]

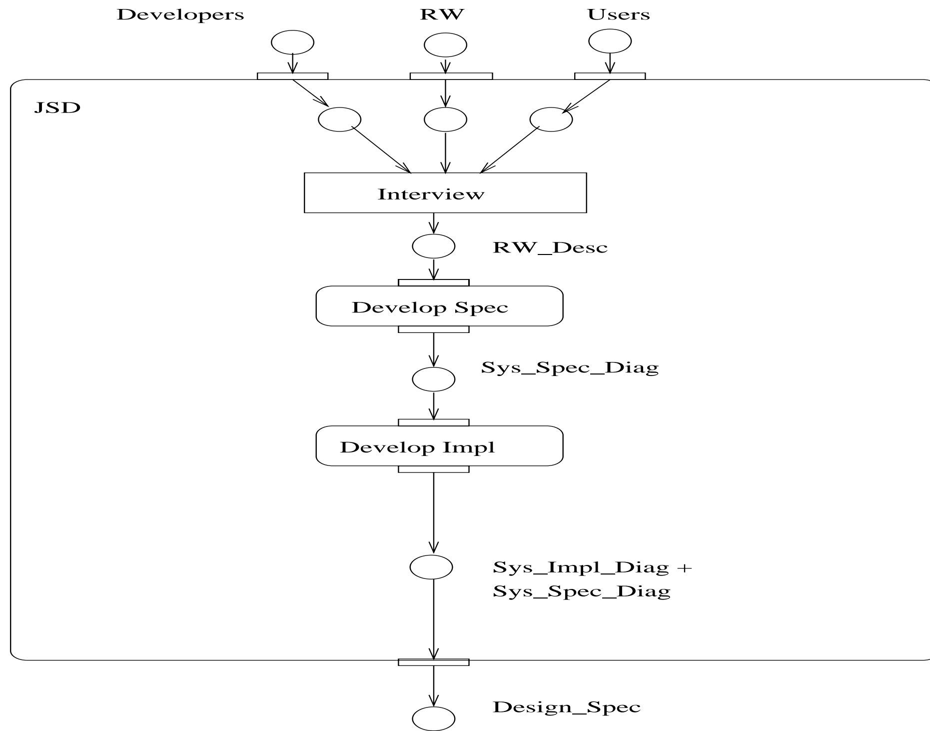

Figure 3: JSD Level 1

BSM features in turn.

3.2.1 Identify Objects

Identify Objects is a feature of BSM. It is a part of the Analysis of Requirements activity (cf. Appendix A.2), and it performs identification of objects in the system being designed.

- $\boldsymbol{Extract_Features_{HFSP,BSM}}$

  We evaluated the function

  $Extract_Features_{HFSP,BSM}(JSD_Model_{HFSP})$ to yield $JSD_FS_{BSM,HFSP}$

  This model of JSD in HFSP does not mention identification of objects (there is no function whose name would imply that it deals with identification of objects), but it mentions entities and functions on them. The notion of an entity in JSD seems to be very close to the notion of an object in BSM because both seem to play the role of autonomous units into which a system is decomposed. The HFSP model of JSD contains a function called Identify_Entity which seems to support the "Identify Objects" feature of BSM. There is no other component of the model that seems to refer to this BSM feature.

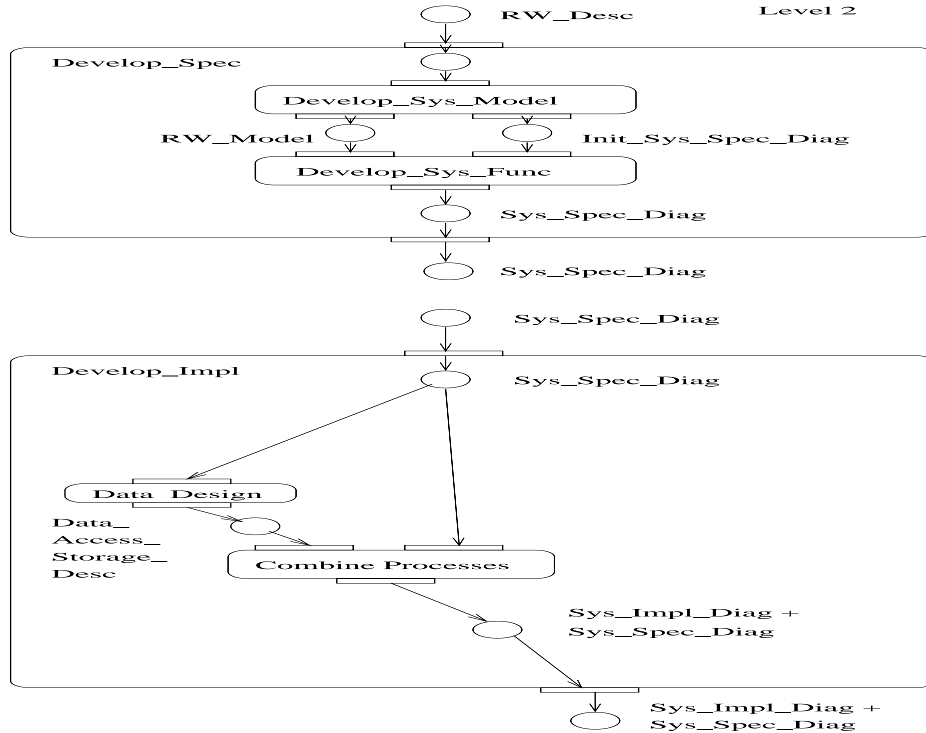

Figure 4: JSD Level 2

We then evaluated the function

$Extract_Features_{HFSP,BSM}(BOOD_Model_{HFSP})$ to yield $BOOD_FS_{BSM,HFSP}$

This model of BOOD in HFSP contains a function called Identify_Objects which obviously supports this feature of BSM (it so happened that the name of this BOOD component coincides with the name of a BSM feature). There is no other component of the model that seems to refer to this class.

We then evaluated the function

$Extract_Features_{HFSP,BSM}(JSD_Model_{SLANG})$ to yield $JSD_FS_{BSM,SLANG}$

This model of JSD in SLANG contains the Identify_Entity transition (Fig. 7). By the above considerations it seems to be an "Identify Objects" feature of BSM.

Finally, we evaluated the function

$Extract_Features_{HFSP,BSM}(BOOD_Model_{SLANG})$ to yield $BOOD_FS_{BSM,SLANG}$

This model of BOOD in SLANG contains the Identify_Object transition at level 1 in Fig. 8 which seems to be a function that supports the "Identify Object" feature.

Thus both HFSP and SLANG identify the same components of JSD and BOOD that support the "Identify Objects" feature of BSM. Now let us compare how effectively the two MFs

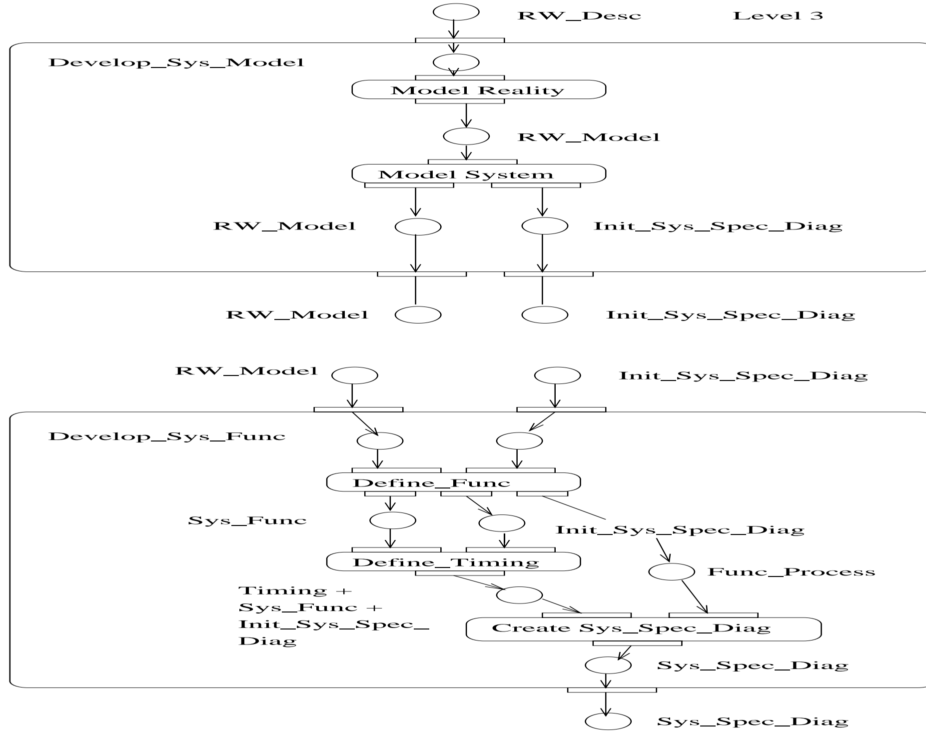

Figure 5: JSD Level 3

support the comparison of the semantics of how these components support this BSM feature.

- $\boldsymbol{Feature_Comparator_{BSM}}$

  We began by evaluating

  $Feature_Comparator_{BSM}(JSD_FS_{BSM,HFSP}, BOOD_FS_{BSM,HFSP})$
  to yield $JSD_BOOD_COMP_{BSM,HFSP}$

  From the model of JSD we can see that the Identify_Entity function produces the artifact called Entity_List from the nouns found in the Real World Description of the system. According to the decomposition it is on the sixth level and depends on two functions, Identify_Noun and Select_Entity, and the following artifacts: Real_World_Desc (the system requirements document) and Action_List.

  The model of BOOD shows us that the Identify_Object function is on the first level of decomposition and hence depends on many more functions than its JSD counterpart. Namely, it depends on the following functions: Identify_Nouns, Identify_Concrete_Objects, Identify_Abstract_Objects, Identify_Server, Identify_Agent, Identify_Actor, Identify_Class, and Identify_Attributes. Judging from its constituent functions, BOOD's Identify_Object also determines objects by finding nouns in the system requirements document (Req_Spec artifact). But there are many more different kinds of objects that are identified. So while the output of

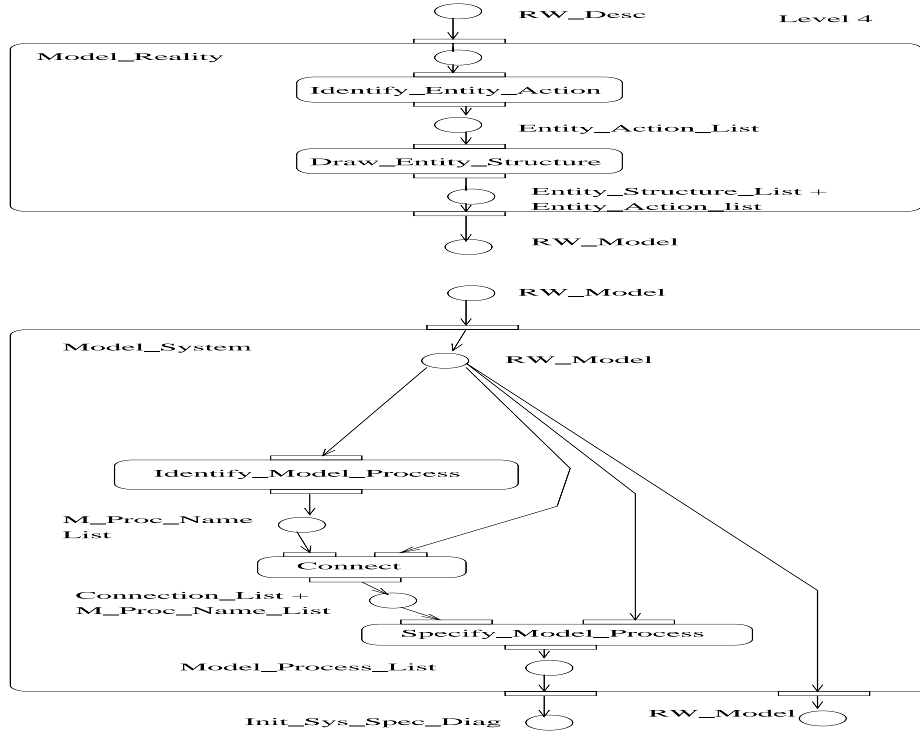

Figure 6: JSD Level 4

JSD's Identify_Entity seems to be a list of objects, the output of BOOD's Identify_Objects is a union of many lists (one per each kind of object). In addition, one can see from the BOOD model that objects in BOOD are also assigned attributes, that is, the objects are described, while the JSD model does not indicate any possible description of its entities.

We then evaluated

$Feature_Comparator_{BSM}(JSD_FS_{BSM,SLANG}, BOOD_FS_{BSM,SLANG})$
to yield $JSD_BOOD_COMP_{BSM,SLANG}$

The same comparison results were obtained from studying the JSD and BOOD SLANG models. Transitions and places of SLANG diagrams seem to map directly to functions and input and output lists of HFSP. For instance, the functions Develop_System_Model and Develop_System_Func found on the second level of the HFSP model of JSD (Appendix A.1) correspond exactly to Develop_Sys_Model and Develop_Sys_Func transitions in the SLANG model of JSD.

These observations are the basis for our determination of the evaluation of

$Compare_Differences_{BSM}(JSD_BOOD_COMP_{BSM,HFSP}, JSD_BOOD_COMP_{BSM,SLANG})$

We concluded that both MFs support reasoning and observations that turn out to be the same. In retrospect this is not surprising as analysis of the "Identify Objects" feature seems

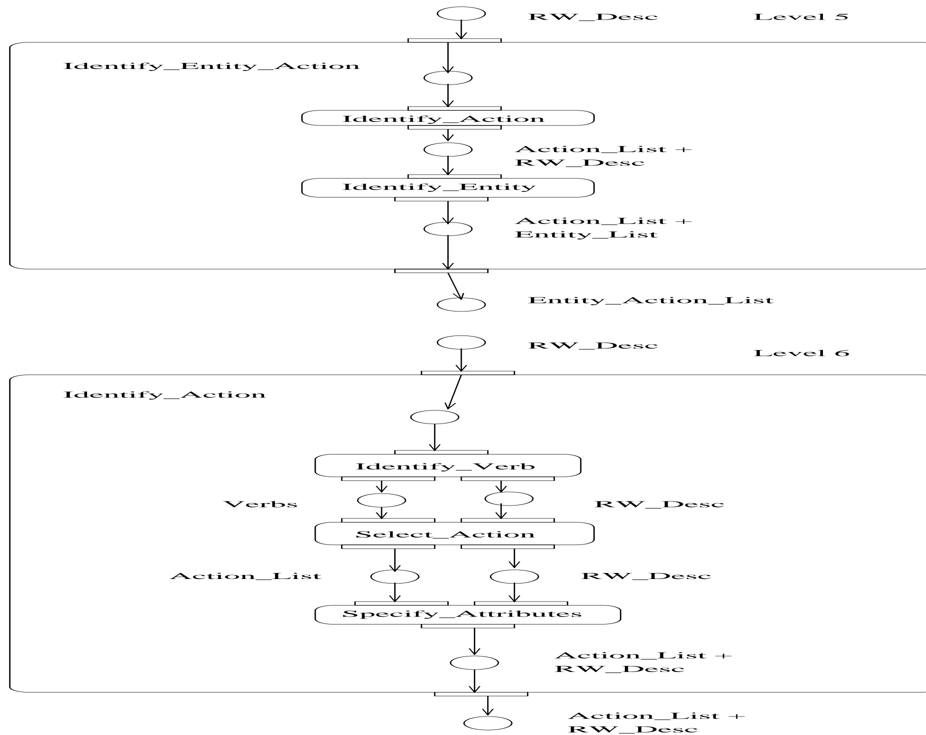

Figure 7: JSD Level 5, 6

to require largely capabilities for functional decomposition and dataflow specification. In these aspects both SLANG and HFSP offer essentially equivalent capabilities. It is worth noting, however, that our observation that BOOD's object list is attributed is easier to see from the HFSP model, which supports artifact decomposition, than from the SLANG model, which does not. The conclusion seems rather easily reached from SLANG-based models, but it is a less direct result. This suggests that it should be particularly interesting to compare comparisons of artifact related features.

3.2.2 Attributes

Thus, let us compare the functioning of JSD and BOOD relative to the "Attribute" feature of BSM.

- $\boldsymbol{Extract_Features_{HFSP,BSM}}$

  We first examine

  - $JSD_FS_{BSM,HFSP}$

    This HFSP model of JSD does not mention Attributes as artifacts directly, but there is a function called Specify_Attributes on the sixth level (Appendix A.1). The output of

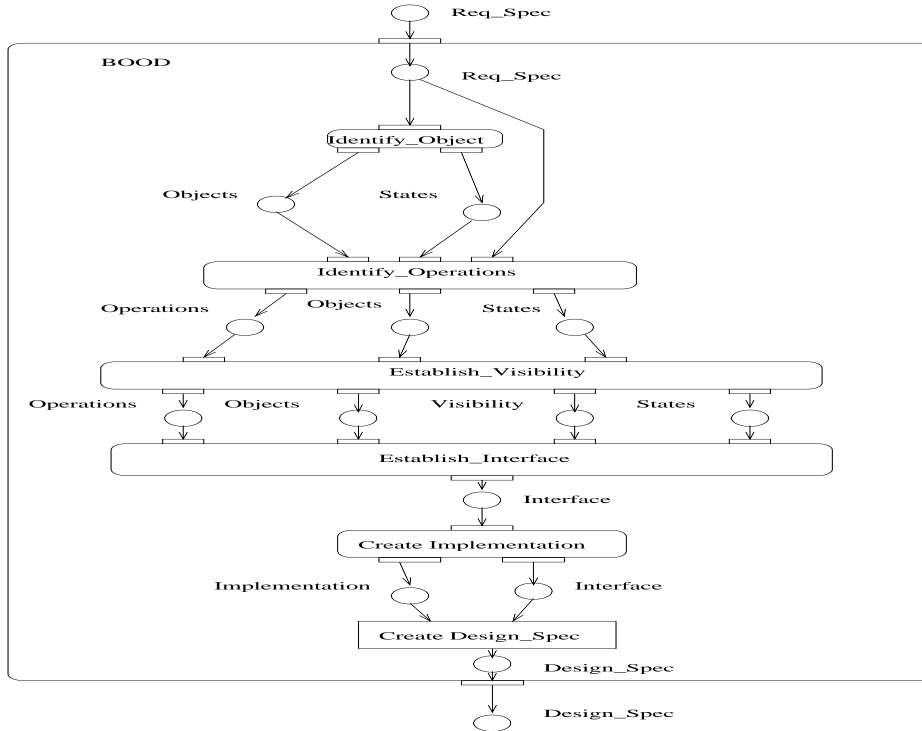

Figure 8: BOOD Level 1

this function is Action_List, obviously augmented with attributes, which was also one of the inputs without attributes. Now, keeping in mind that Action_List has attributes added to it at the sixth level of decomposition, we can say that attributes are part of Design Description (the final product of the SDM shown at level 1). The model also shows that the attributes describe actions of the objects identified in the system being developed because the Specify_Attributes function (sixth level, Appendix A.1) takes as input Real_World_Desc and Action_List. That is, the mentions of actions are found in the real world descriptions and are assigned attributes based on the information presented in the real world description. So, we can say that JSD supports the feature of identifying attributes. This is inferred indirectly from the HFSP model.

Next we examine

– $BOOD_FS_{BSM,HFSP}$

The HFSP model of BOOD mentions attributes as a product of the Specify_Attr function (Appendix A.2, second level, line e2) and shows that Attributes are part of Interface (same figure, line e6) which is part of Design Specification (the product of the whole SDM, line a6). In addition, the functions' I/O lists suggest that States, Modules, and Objects are assigned attributes and are hence described indirectly (this can be seen in line e2 which shows that States and Modules are assigned Attributes, and in line b8

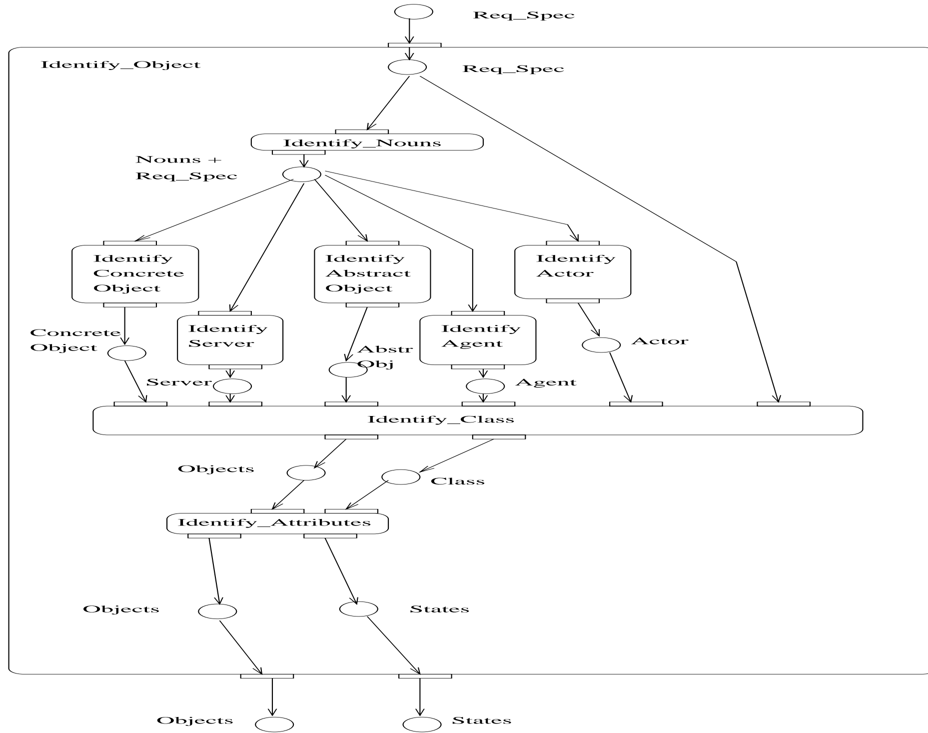

Figure 9: BOOD Level 2

which shows that Objects are assigned attributes).

Now we examine

- $JSD_FS_{BSM,SLANG}$ and $BOOD_FS_{BSM,SLANG}$

  Virtually the same inferences can be drawn from the SLANG models because there is again an exact correspondence between HFSP functions and SLANG transitions and between HFSP I/O structures and SLANG places. For instance, the Specify_Attributes function of the HFSP model of JSD corresponds to the Specify_Attributes transition in the SLANG model (level 6, Fig. 7) which also shows that the Action List artifact is both an input and an output of this transition. The SLANG model of BOOD contains the Identify_Attributes transition (Fig. 9) corresponding to the Identify_Attributes function in the HFSP model of BOOD. The Specify_Attributes transition corresponding to the Specify_Attributes function in the HFSP model can be found in Fig. 12. Using the SLANG models we can arrive at analogous results, but we have to track the input and output places of the transitions to see how artifacts are possibly modified (for example, to see that Action_List does not contain attributes before the Specify_Attributes transition in the SLANG model of JSD while after this transition it does).

We then executed the third functional step of CDM, namely:

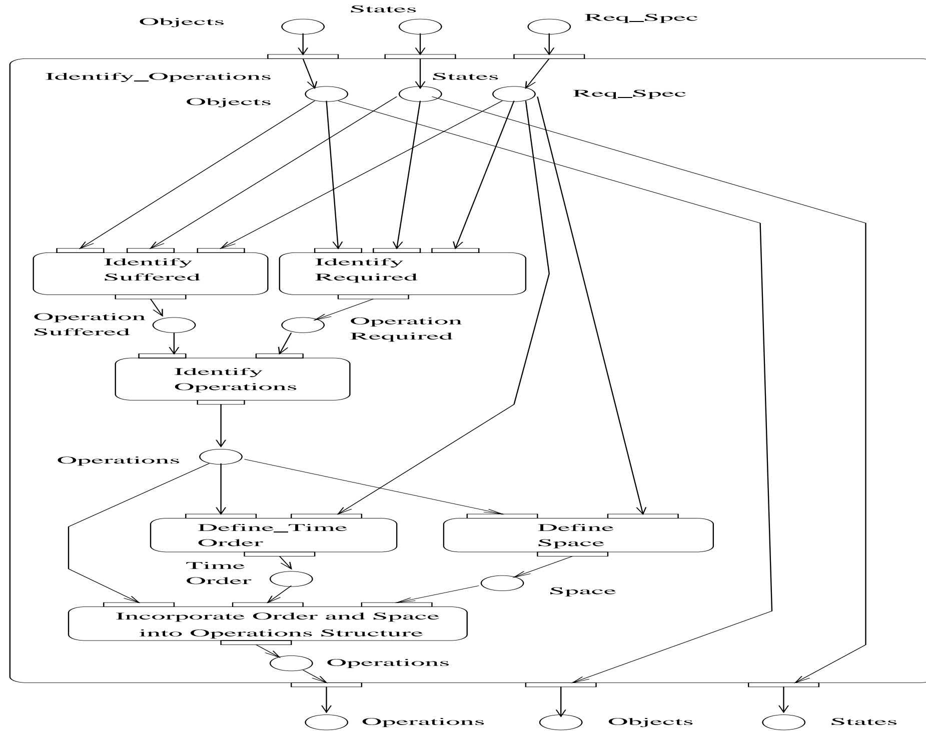

Figure 10: BOOD Level 2

- ***Feature_Comparator***$_{BSM}$

  Now that we have found the components of the two models that deal with the Attributes feature of BSM we can compare them. As in Sec. 3.2.1, this entails examining first $JSD_BOOD_COMP_{BSM,HFSP}$, obtained as the result of the function $Feature_Comparator_{BSM}(JSD_FS_{BSM,HFSP}, BOOD_FS_{BSM,HFSP})$

  In JSD, Attributes are part of the Action_List as determined during classification and they most likely describe the actions of the objects mentioned in the real world description. Eventually, attributes are part of the final product. This can be determined from the HFSP model of JSD (Appendix A.1) in the following way:

  1. line i3: The Specify_Attributes function adds attributes to the Action_List artifact which is the output of the Identify_Action function
  2. line g3: This artifact decomposition statement shows that Action_List is a component of the Entity_Action_List which is the output of the Identify_Entity_Action function
  3. line e2: The Draw_Entity_Structure function incorporates the Entity_Action_List artifact into the Entity_Structure_List artifact
  4. line e3: This artifact decomposition statement shows that the Entity_Structure_List artifact is the Real_World_Model artifact which is the output of the Model_Reality activity

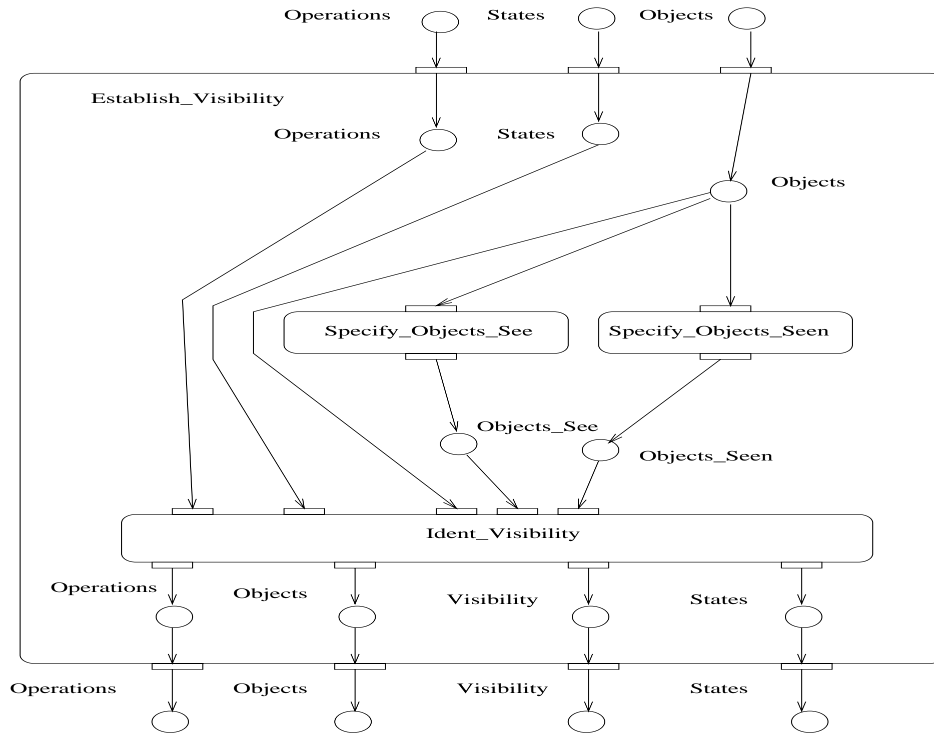

Figure 11: BOOD Level 2

5. line c2: The Model_System function incorporates the Real_World_Model artifact into the Init_System_Spec_Diagram artifact which is the output of the Develop_System_Model function
6. line b2: The function Develop_System_Func incorporates the Init_System_Spec_Diagram artifact into the System_Spec_Diagram artifact which is the output of the Develop_Spec function
7. line a4: This artifact decomposition statement shows that the System_Spec_Diagram artifact is part of Design_Spec which is the product of JSD

The HFSP model of JSD is not detailed enough to show the structure of the Action_List, so it is not clear what the structure of the artifact containing the attributes is. Probably it is a table (implemented either as a list or an array) containing the names of the action attributes and their corresponding values.

In BOOD, not only Objects, but also States and Modules, are assigned attributes. Attributes are part of the final product. This can be seen from the HFSP model of BOOD (Appendix A.2) in the following way: Attributes of Objects are in the final product because of

1. line b8: The Identify_Attributes function produces the States artifact from the Objects artifact and, by this, apparently incorporates the attributes of objects into the States

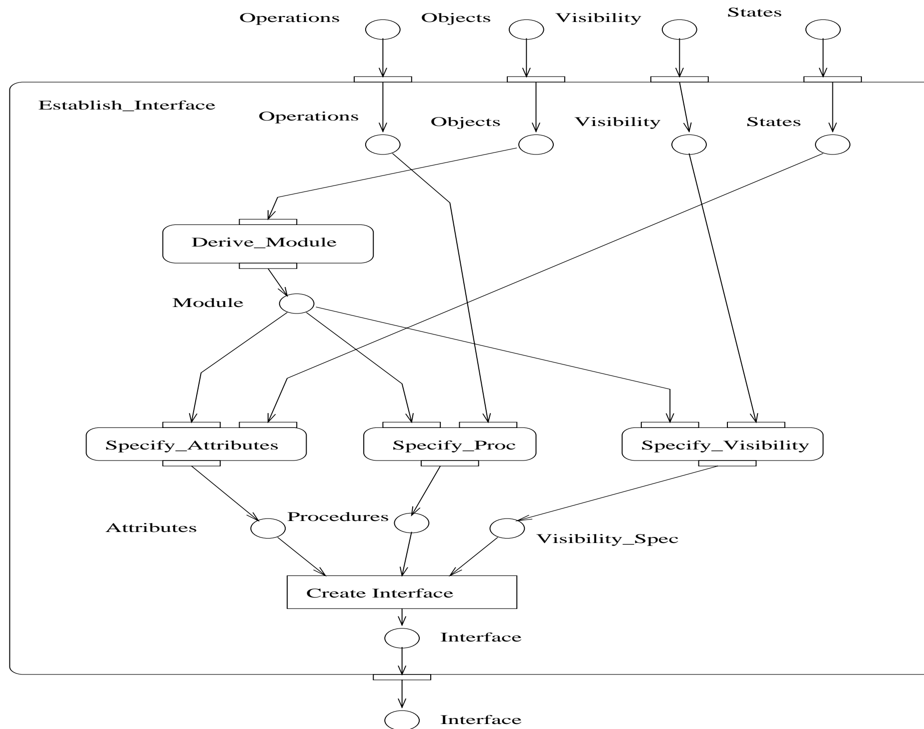

Figure 12: BOOD Level 2

artifact which is the output of the Identify_Object function

2. line a4: The Establish_Interface function incorporates the States artifact into the Interface artifact

3. line a6: This artifact decomposition statement shows that the Interface artifact is part of Design_Spec which is the product of BOOD

Attributes of States and Modules are in the final product because of

1. line e2: The Specify_Attr function produces the Attributes artifact from the States and Module artifacts
2. line e6: This artifact decomposition statement shows that the Attributes artifact is part of the Interface artifact which is the output of the Establish_Interface function
3. line a6: This artifact decomposition statement shows that the Interface artifact is part of Design_Spec which is the product of BOOD

The name of the Identify_Attributes function suggests that it just identifies the names of the attributes but not their values, while the name of the Specify_Attr function suggests that it assigns values to the attributes. The model also suggests that the names identified in

the Identify_Attributes function are specified in the Specify_Attr function because the States artifact is an output of the former and then input to the latter.

It seems that the notions of Objects are similar in JSD and BOOD. But from the HFSP models it seems that BOOD is more meticulous about describing entities in terms of their attributes. To be sure of this it seemed advisable to find correspondences of BOOD's States and Modules in JSD, if any.

Following CDM, we next examined $JSD_BOOD_COMP_{BSM,SLANG}$, the output of function $Feature_Comparator_{BSM}(JSD_FS_{BSM,SLANG}, BOOD_FS_{BSM,SLANG})$.

The SLANG model of JSD shows that the Action_List artifact describes the actions of the objects found in the real world description. It is more difficult to see this in the SLANG model because SLANG does not have an artifact decomposition capability. One has to follow a certain path of places in the SLANG diagrams to see whether a certain artifact is part of another artifact. In the SLANG diagrams it can be only assumed that the artifact in the input place of a transition is put into the output place (unchanged) while HFSP states the artifact decomposition explicitly. The results concerning the relationship of the Identify_Attributes and Specify_Attr functions in BOOD can be obtained from the SLANG model because the reasoning involves only the names of transitions and input/output places.

3.2.3 Concurrency

The final BSM feature we discuss is concurrency. This feature is mentioned explicitly in Brinkkemper et al.'s supermethod in that two features, Concurrency between Objects and Concurrency within Objects deal with concurrency. In [4] (p. 129) it is stated that "Concurrency is a property for describing Systems in which more Objects operate in parallel". Thus, those SDM components that allow design of concurrent systems should be classified as relating to BSM's notion of concurrency. These components seemed to include:

- the artifacts containing information about communication between entities of the software system designed and the entities themselves
- the activities operating on such artifacts

Let us see which SDM components satisfy these criteria.

- $\boldsymbol{Extract_Features_{HFSP,BSM}}$

  The components of $JSD_FS_{BSM,HFSP}$ that seem to be directly involved with support of the design of concurrent systems are (cf. Appendix A.1):

  - line f5: The Connection artifact, which is composed of the State_Vector and Data_Stream artifacts

* line f2: The Connect function, which produces the Connection_List artifact out of the Real_World_Model and M_Proc_Name_List artifacts. The relationship between the Connection_List and Connection artifacts is not clear from the model, but, apparently, the Connection_List artifact is a list of the Connection artifacts
* The Entity_Structure_List artifact is the Real_World_Model artifact (line e3). The Real_World_Model artifact is used to produce the M_Proc_Name_List (line f1)
  These JSD components seem to be relevant to the Concurrency between Objects feature. No components were found which would clearly be seen as being relevant to the Concurrency within Objects feature.

The following components of $BOOD_FS_{BSM,HFSP}$ seem relevant to the Concurrency between Objects feature of BSM (cf. Appendix A.2):

* line b9: The Objects artifact
* line e6: The Interface artifact which is composed of the Attributes, Procedure, and Visibility_Spec artifacts
* line e: The Establish_Interface function (i.e., all the functions into which it is decomposed)
  No components can be easily identified as part of the Concurrency within Objects feature.

$JSD_FS_{BSM,SLANG}$ shows that the following components seem to support the Concurrency between Objects feature:

* The Connect transition (level 4, Fig. 6)
* The Connection_List and RW_Model artifacts

Since SLANG does not support artifact decomposition it is not possible to see that the Connection artifact is composed of the State_Vector and Data_Stream artifacts. Also the relationship between the RW_Model artifact and Entity_Structure_List is not clear. From Fig. 6 one can see that the RW_Model artifact seems to be composed of the Entity_Structure_List and Entity_Action_List artifacts. The HFSP model clearly states that the Entity_Structure_List artifact is the Real_World_Model artifact.

$BOOD_FS_{BSM,SLANG}$ indicates that the following components seem to support the Concurrency between Objects feature:

* The Establish_Interface transition (level 1, Fig. 8)
* The Interface place (artifact) (the same figure)
* The Objects place (artifact) (the same figure)

No components can be identified as being relevant to the Concurrency within Objects feature. The SLANG model fails to demonstrate that the Interface artifact is composed of the Attributes, Procedure, and Visibility_Spec artifacts.

Now let us compare the identified components.

- ***Feature_Comparator*** $_{BSM}$

  In producing $JSD_BOOD_COMP_{BSM,HFSP}$ we see that $JSD_FS_{BSM,HFSP}$ shows that the Real_World_Model artifact keeps information about the objects and their structure. Objects do not seem to be further subdivided into different classes. The Connect activity takes the description of objects (the Real_World_model) and produces the Connection_List artifact, which has Connection artifacts as its elements. $JSD_FS_{BSM,HFSP}$ also shows that the Connection artifact may be either a State_Vector or Data_Stream artifact which indicates that JSD directly addresses the issue of communication modeling between objects. $BOOD_FS_{BSM,HFSP}$ shows that BOOD uses the Objects artifact to keep information about the objects in the system. BOOD does not seem to contain information about the structure of the objects as JSD does. On the other hand, BOOD distinguishes many classes of Objects. $JSD_FS_{BSM,HFSP}$ shows that the Objects artifact is comprised of the Concrete_Object, Abstract_Object, Class, Agent, Actor, and Server artifacts (Appendix A.2, line b9). The last three artifacts indicate that BOOD also distinguishes the roles played by the objects. The communication between objects is specified in the Interface artifact produced by the Establish_Interface activity. BOOD does not seem to model communication at as low a level as JSD does with State_Vector and Data_Stream. $BOOD_FS_{BSM,HFSP}$ shows that the Interface artifact is composed of the Attributes, Procedure, and Visibility_Spec attributes.

  In producing $JSD_BOOD_COMP_{BSM,SLANG}$ we see that $JSD_FS_{BSM,SLANG}$ allows us to see that communication between objects is described in the Connection_List artifact, the information about objects is kept in the RW_Model artifact, and the Connect activity produces the Connection_List. $JSD_FS_{BSM,SLANG}$ does not contain the decomposition of the Connection_List artifact, so it is not possible to see the communication that JSD models through the State_Vector and Data_Stream artifacts.

  $BOOD_FS_{BSM,SLANG}$ shows that BOOD keeps information about objects in the Objects artifact, describes communication in the Interface artifact, and uses the Establish_Interface activity to create the communication description. The SLANG diagram shown in Fig. 9 implicitly shows that the Objects artifact is probably composed of the Concrete_Object, Server, Abstract_Object, Agent, and Actor artifacts while $BOOD_FS_{BSM,HFSP}$ indicates it explicitly in an artifact decomposition statement. Considering this, BOOD seems to have a more developed object model while JSD seems to be more specific about communication between objects.

4 Conclusions

As has been noted above, we were struck by the fact that the comparison results obtained based upon HFSP models were generally very much the same as the results obtained using SLANG models. Both HFSP and SLANG feature the use of hierarchical functional decomposition at their central semantic feature, and both also place major emphasis upon the specification of dataflow. Thus, strong similarities in comparison results were not unexpected. But the degree of closeness of the comparison results to each other was beyond our expectations.

We noted initially that HFSP offers support for the specification of artifact decomposition, while SLANG does not, and we expected that this would enable HFSP to be noticeably more effective in making distinctions between how BOOD and JSD dealt with issues involving design artifacts. Indeed we did note that HFSP facilitated the detection of some details in the handling of attributes. On the other hand, we were surprised to see how readily these details were indirectly inferred from SLANG models. Thus, the advantage we had expected from the use of HFSP did not turn out to be as great as we had expected.

Other differences between HFSP and SLANG also did not turn out to be pivotal. SLANG's visual representations by means of Petri Nets seemed comfortable and appealing, but we did not sense that it fostered clearer mental images in the mind of the analyst than the images projected from HFSP specifications. SLANG also supports stronger specification of concurrency. But since in our experiment we compared how well SDMs support development of concurrent systems as opposed to the level of concurrency in SDMs themselves, this particular advantage of BOOD was not highlighted.

It is certainly true that comparison over just the three features described in this paper is hardly a sufficient basis for conclusive findings about the importance of the modeling formalism, but our informal considerations of a range of other SDM features in BSM seemed to all yield the same conclusion. Namely, that HFSP and SLANG tended to support very much the same comparison results.

It does seem important to note that our comparison results are ultimately based upon a less firm foundation than we would like, in that they are based upon our informal interpretations of what is meant by the various SDM features in BSM. As Brinkkemper, et. al. in [4] and Song and Osterweil in [10] described SDM features informally, there is no definitive way to be sure that the features we selected for comparison in this work are precisely the right features. We believe that our selections are the appropriate ones, but we acknowledge that the validity of this work can be attacked due to possible disagreements about whether the SDM components we selected as being supportive of the various features are the right ones. This difficulty can only be remedied when CS features are defined rigorously. This is an extension of this work that seems most important to pursue.

In addition, we noted that identification of appropriate components to support the various features was hampered from time to time by inadequacies in the actual SDM models we were studying. In some cases it turned out that the models themselves lacked needed details. This was not the fault of the modeling formalisms, but rather of the modeler using the formalisms. In some cases we felt that the comparisons would have been even more similar to each other had the modeler been more effective. Thus, another conclusion we draw is that the accuracy of the model, while strongly affected by the modeling formalism used, is at least as critically affected by the skill of the modeler using the formalism.

5 Future Work.

There are clearly a number of key directions in which this work should be continued. Certainly there is now strong motivation to continue comparing comparisons over a far wider range of BSM features than just the three described here. This work is necessary to either confirm and undermine the conjecture expressed here that the MF may not be as critically important as had originally been anticipated.

Especially if that conjecture holds up to further experimental scrutiny, then it seems most important to consider the effect of MF's that are more fundamentally different from each other. As has been observed, we purposely selected two formalisms having strong fundamental similarities. These similarities seem to have been sufficient to assure essentially similar comparison results. It now seems important to carry out this same experiment using modeling formalisms that are more radically different. The use of a rule-based formalism such as Prolog or Marvel, for example, seems to represent a next logical step in this sort of investigation.

As noted above, it also seems important to set this line of research on a more solid foundation of rigor by using formalisms to define the CS comparison framework and the features within in. A formalism capturing the structure of comparison topics also will be very helpful. The comparison topics of our experiment were not rigorously defined (i.e., activities, concepts, techniques, additional properties of methods). Such a definition would certainly make the comparison results more objective, since there would be less room for ambiguity. In addition, the comparer would have clear guidelines concerning the actual comparison of two SDM components and the comparison job would become easier and more repeatable. Furthermore, it would be beneficial to introduce a more formal measure between SDMs based on the CS used and comparison topics used to compare features identified by the CS.

In closing, we feel it is important to emphasize that we feel that this experiment is strongly encouraging in that it does seem to indicate that rigorous, reproducible comparison of SDM's is quite feasible. This line of research was undertaken in reaction to a long string of SDM comparison work that was essentially completely informal, offering no basis for scientific validation through reproducible experimentation. It was our hope that SDM comparison could be made rigorous, semantically well-founded, and reproducible through the use of formally defined comparison processes (such as CDM), comparison schemas (such as BF and BSM), and semantically well-based modeling formalisms (such as HFSP and SLANG). This work continues to provide evidence that this sort of rigor and reproducibility is possible. It goes further than earlier work, however, in suggesting that the comparisons attempted may be far less sensitive to differences in choices of modeling formalism. That will make this sort of comparison far more amenable to community participation than if all comparisons needed to be made based upon one single, agreed-upon formalism.

6 Acknowledgments

Xiping Song provided invaluable help and suggestions throughout this study.

A Appendix

A.1 Model of JSD in HFSP

(a) JSD$(Real_World|Design_Spec) \Rightarrow$
(1) $Develop_Spec(Real_World_Desc|System_Spec_Diagram)$
(2) $Develop_Impl(System_Spec_Diagram|System_Impl_Diagram)$
(3) **Where** $Real_World_Desc = Interview(Users, Developers, Real_World)$,
(4) $Design_Spec = union(System_Spec_Diagram, System_Impl_Diagram)$;

Second_level:
(b) Develop_Spec$(Real_World_Desc|System_Spec_Diagram) \Rightarrow$
(1) $Develop_System_Model(Real_World_Desc|Real_World_Model, Init_System_Spec_Diagram)$
(2) $Develop_System_Func(Real_World_Model, Init_System_Spec_Diagram|System_Spec_Diagram)$;

Third_level:
(c) Develop_System_Model$(Real_World_Desc|Real_World_Model, Init_System_Spec_Diagram) \Rightarrow$
(1) $Model_Reality(Real_World_Desc|Real_World_Model)$
(2) $Model_System(Real_World_Model|Init_System_Spec_Diagram)$;

(d) Develop_System_Func$(Real_World_Model, Init_System_Spec_Diagram|System_Spec_Diagram) \Rightarrow$
(1) $Define_Func(Real_World_Model, Init_System_Spec_Diagram|System_Function, Function_Process)$
(2) $Define_Timing(Init_System_Spec_Diagram, System_Function|Timing)$
(3) **Where** $System_Spec_Diagram =$
$is_composed_of(Init_System_Spec_Diagram, System_Function, Function_Process, Timing)$;

Fourth_level:
(e) Model_Reality$(Real_World_Desc|Real_World_Model) \Rightarrow$
(1) $Identify_Entity_Action(Real_World_Desc|Entity_Action_List)$
(2) $Draw_Entity_Structure(Entity_Action_List|Entity_Structure_List)$
(3) **Where** $Real_World_Model = is(Entity_Structure_List)$,
(4) $Real_World_Process = is(Entity_Structure)$;

(f) Model_System$(Real_World_Model|Init_System_Spec_Diagram) \Rightarrow$
(1) $Identify_Model_Process(Real_World_Model|M_Proc_Name_List)$;
(2) $Connect(Real_World_Model, M_Proc_Name_List|Connection_List)$
(3) $Specify_Model_Process(Connection_List, Real_World_Model, M_Proc_Name_List|Model_Process_list)$
(4) **Where** $Init_System_Spec_Diagram = is(Model_Process_List)$;
(5) $Connection = is(State_Vector)$ or $is(Data_Stream)$;

Fifth_level Decomposition:
(g) Identify_Entity_Action$(Real_World_Desc|Entity_Action_List) \Rightarrow$
(1) $Identify_Action(Real_World_Desc|Action_List)$
(2) $Identify_Entity(Real_World_Desc, Action_List|Entity_List)$
(3) **Where** $Entity_Action_List = union(Action_List, Entity_List)$;

Sixth_level Decomposition:
(i) **Identify_Action**$(Real_World_Desc|Action_List) \Rightarrow$
(1) $Identify_Verb(Real_World_Desc|Verbs)$
(2) $Select_Action(Real_World_Desc, Verbs, Entity_List|Action_List)$
(3) $Specify_Attributes(Real_World_Desc, Action_List|Action_List)$;

(h) **Identify_Entity**$(Real_World_Desc, Action_List|Entity_List) \Rightarrow$
(1) $Identify_Noun(Real_World_Desc|Nouns)$
(2) $Select_Entity(Real_World_Desc, Nouns, Action_List|Entity_List)$;

A.2 Model of BOOD in HFSP

(a) BOOD($Req_Spec|Design_Spec$) $\Rightarrow$
(1) $Identify_Object(Req_Spec|Objects, States)$
(2) $Identify_Operations(Req_Spec, Objects, States|Operation)$
(3) $Establish_Visibility(Req_Spec, Objects, States, Operation|Visibility)$
(4) $Establish_Interface(Visibility, Objects, States, Operation|Interface)$
(5) $Establish_Implementation(Interface|Implementation)$
(6) **Where** $Design_Spec = is_composed_of(Interface, Implementation)$;

Second Level:
(b) Identify_Object($Req_Spec|Objects, States$) $\Rightarrow$
(1) $Identify_Nouns(Req_Spec|Nouns)$
(2) $Identify_Concrete_Object(Req_Spec, Nouns|Concrete_Object)$
(3) $Identify_Abstract_Object(Req_Spec, Nouns|Abstract_Object)$
(4) $Identify_Server(Req_Spec, Nouns|Server)$
(5) $Identify_Agent(Req_Spec, Nouns|Agent)$
(6) $Identify_Actor(Req_Spec, Nouns|Actor)$
(7) $Identify_Class(Req_Spec, Agent, Server, Actor, Concrete_Object, Abstract_Object|Class)$
(8) $Identify_Attributes(Objects|States)$
(9) **Where** $Objects = union(Concrete_Object, Abstract_Object, Class, Agent, Actor, Server)$

(c) Identify_Operation($Req_Spec, Object, States|Operation$) $\Rightarrow$
(1) $Identify_Suffered(Req_Spec, Object, States|Operation_Suffered)$
(2) $Identify_Required(Req_Spec, Object, States|Operation_Required)$
(3) $Defining_Time_Order(Req_Spec, Operation|Time_Order)$
(4) $Defining_Space(Req_Spec, Operation|Space)$
(5) **Where** $Operation = union(Operation_Suffered, Operation_Required)$

(d) Establish_Visibility($Req_Spec, Objects, States, Operation|Visibility$) $\Rightarrow$
(1) $Specify_Object_See(Objects|Objects_See)$
(2) $Specify_Object_Seen(Objects|Object_Seen)$
(3) **Where** $Visibility = union(Objects_See, Object_Seen)$

(e) Establish_Interface($Visibility, Object, States, Operations|Interface$) $\Rightarrow$
(1) $Derive_Module(Object|Module)$
(2) $Specify_Attr(States, Module|Attributes)$
(3) $Specify_Proc(Operations, Module|Procedures)$
(4) $Specify_Visibility(Visibility, Module|Visibility_Spec)$
(5) **Where** $Subsystem = is_in_term_of(Module)$,
(6) $Interface = is_composed_of(Attributes, Procedure, Visibility_Spec)$;

B Appendix

B.1 Superclasses of CS (activities of the supermethod)

	Analysis	Design	
1.1	Analysis of Requirements	2.1	Initial Design
1.2	Analyze Objects and Classes	2.2	Class and Class Structure Design
1.3	Analyze Relationships	2.3	Operations Design
1.4	Analyze Attributes	2.4	Relationship/Attribute Design
1.5	Partitioning	2.5	Human Interaction Design
1.6	Dynamic Behavior	2.6	Data Management Design
1.7	Analyze Functional behavior/Operations	2.7	System Design
1.8	Analyze Communication	2.8	Validation

B.2 Classes of CS (subactivities of the supermethod)

1.1.1 Understand requirements specification
1.2.1 Identify Objects
1.2.1a Identify Classes
1.2.2 Name Classes
1.2.2a Name Objects
1.2.3 Describe Classes and Objects
1.2.4 Apply guidelines to control Classes and Objects
1.2.5 Identify Abstract Classes
1.2.6 Search for missing Classes
1.3.1 Identify Inheritance relationships
1.3.2 Identify part-of relationships
1.3.3 Identify Multiple Structures
1.3.4 Identify Associations
1.3.5 Identify Abstract/Concrete Classes
1.3.6 Identify other relationships
1.3.7 Describe associative objects
1.3.8 Check Associations
1.3.9 Check inheritance relationship
1.4.1 Identify Attributes
2.5.2 Describe humans and task scenarios
1.4.2 Position Attributes
1.4.3 Check Attributes
1.4.4 Describe Attributes
1.5.1 Select Subsystems
1.5.2 Refine Subsystems
1.6.1 Prepare Scenarios
1.6.2 Identify Events
1.6.3 Build Event-Flow Diagrams
1.6.4 Match events between objects
1.6.5 Build State Diagrams
1.6.11 Identify Operation
1.7.2 Build Data-Flow Diagrams
1.7.3 Describe functions
1.8.1 Identify collaborations/message connections
2.2.7 Support Data Man. Component
2.2.8 Add lower level components
2.3.2 Choose Algorithms
2.3.3 Choose data-structures
2.5.1 Classify the humans
2.7.8 Allocate programs to processors

References

[1] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Luigi Lavazza. SPADE: An Environment for Software Process Analysis, Design, and Enactment. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, chapter 9, pages 223–248. Research Studies Press, Ltd., Taunton, Somerset, England, 1994.

[2] Grady Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, February 1986.

[3] John R. Cameron. An Overview of JSD. *IEEE Transactions on Software Engineering*, SE-12(2):222–240, February 1986.

[4] Sjaak Brinkkemper Geert van den Goor, Shuguang Hong. A Comparison of Six Object-Oriented Analysis and Design Methods. Technical report, University of Twente, Enschede, the Netherlands, 1992.

[5] Takuya Katayama. A Hierarchical and Functional Software Process Description and its Enaction. In *Proceedings of the Eleventh International Conference of Software Engineering*, pages 343–353, 1989.

[6] Leon J. Osterweil. Software Process Interpretation and Software Environments. Technical Report CU–CS–324–86, Department of Computer Science, University of Colorado, Boulder, CO, April 1986.

[7] Xiping Song and Leon Osterweil. Toward Objective, Systematic Design-Method Comparisons. *IEEE Software*, pages 43–53, May 1992.

[8] Xiping Song and Leon J. Osterweil. The Models of the Design Methodologies. Technical Report UCI–91–19, University of California, Irvine, Irvine, CA 92717, 1991.

[9] Xiping Song and Leon J. Osterweil. Engineering Software Design Processes to Guide Process Execution,. Technical Report TR–94–23, University of Massachusetts, Computer Science Department, Amherst, MA, February 1994. Appendix accepted and published in Preprints of the Eighth International Software Proces Workshop.

[10] Xiping Song and Leon J. Osterweil. Experience with an approach to comparing software design methodologies. *IEEE Transactions on Software Engineering*, 20(5):364–384, May 1994.

[11] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. APPL/A: A Language for Software-Process Programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286, July 1995.