1

Rapid #: -4543793

Ariel

IP: 150.135.238.50

Status	Rapid Code	Branch Name	Start Date
Pending	GAT	Main Library	6/28/2011 5:29:28 PM

CALL #: **TA168 .J66**

LOCATION: **GAT :: Main Library :: Main Library, 5th Floor East**

TYPE: Article CC:CCL

JOURNAL TITLE: Journal of systems engineering

USER JOURNAL TITLE: Journal of Systems Engineering

GAT CATALOG TITLE: Journal of systems engineering.

ARTICLE TITLE: Integrating Systems Formalisms: How Object-Oriented Programming Supports Cast for Intelligent System Design

ARTICLE AUTHOR: Zeigler, B.P. and Praehofer, H. and Rozenblit, J.W

VOLUME: 3

ISSUE:

MONTH:

YEAR: 1993

PAGES: 209-219

ISSN: 0938-7706

OCLC #: GAT OCLC #: 27652849

CROSS REFERENCE ID: [TN:957015][ODYSSEY:150.135.238.6/ILL]

VERIFIED:

BORROWER: **AZU :: Main Library**

PATRON: **George Hwang**

PATRON ID: hwangg

PATRON ADDRESS:

PATRON PHONE:

PATRON FAX:

PATRON E-MAIL:

PATRON DEPT:

PATRON STATUS:

PATRON NOTES:

System Date/Time: 6/29/2011 6:54:29 AM MST

J Syst Eng (1993)3:209–219

Journal of Systems Engineering

Integrating Systems Formalisms: How Object-Oriented Programming Supports Cast for Intelligent Systems Design

Bernard P. Zeigler[1], Herbert Praehofer[2] and Jerzy Rozenblit[1]

[1]Department of Electrical and Computer Engineering, The University of Arizona, Tucson, Arizona 85721, USA
[2]Department of Systems Theory and Information Engineering, Institute of Systems Sciences, Johannes Kepler University, A-4040 Linz, Austria

This paper proposes a means to unify the burgeoning variety of modelling and knowledge representation formalisms that are entering into practical use for intelligent systems design. Conceptual frameworks for intelligent system design recognise the critical role of models to structure knowledge representation and utilisation in such systems. However, they do not provide support for employing the great variety of formalisms available for this purpose. We propose principles for unifying the various formalisms within a systems theoretic framework whose implementation is supported by the object-oriented paradigm. In this approach, models can be developed as instances of formalism-based classes of dynamic systems. Moreover, such an approach facilitates combining formalisms so that multiformalism models can be constructed.

Keywords: Combined simulation; Intelligent system modelling; Multiformalism modelling; Object-oriented programming

1. Introduction: Systems, Models and Cast

Systems theory owes its utility to the fact that real systems can obey the same 'system' laws and show similar patterns of behaviour although they are physically very different [1]. This potential isomorpy makes it possible to employ common representations to treat different real systems in a uniform manner [2]. Various systems theories have been developed [3–5] to provide such integrative frameworks. Progress in hardware and software technologies has fostered the development of computer-based tools to implement such systems theory frameworks. The goal of a new field called Computer Aided Systems Theory (CAST) [3,6] is to 'bundle' system theoretical problem solving techniques into user-friendly, easy-to-handle and easy-to-learn packages, thereby increasing their accessibility to the practising engineer.

This paper considers the role of CAST in supporting the design of intelligent systems. The distinguishing issue in designing such systems is how to endow them with the knowledge required to perform their missions. A great variety of modelling and knowledge representation formalisms are entering into practical use for this purpose. To name a few: discrete event dynamic systems [7], fuzzy logic [8] and neural nets [9] are being heavily investigated in the control field, as are genetic algorithms [10], qualitative simulation [11], case-based planning [12,13], reasoning under uncertainty [14] and non-monotonic logic [15] in artificial intelligence.

Conceptual frameworks for intelligent system design [16–18] recognise the critical role of models to structure knowledge representation and utilisation in such systems. However, they do not provide guidance in employing the great variety of formalism just enumerated for this purpose. Intelligent system design requires a methodology for task decomposition, assignment of engines to subtasks and the integration of engines into execution hierarchies matching the task decompositions. From the model-

Received 1 June 1993

Correspondence and offprint requests to: B. P. Zeigler, Department of Electrical and Computer Engineering, The University of Arizona, Tucson, Arizona 85721, USA.

based architecture perspective, the task-performing engines should have two components: an interpreter that is specialised for tasks of a particular type and a model that supplies the interpreter with the specifics of the environment in which the task is to be performed [19]. Such models are expressed in the formalisms that have been enumerated above. However, if the great variety of formalisms is to be marshalled for systems design in this manner, the systems designer must be able to gain access to them in an organised way.

Computer simulation offers an attractive alternative to develop and test intelligent systems in comparison to real test bed environments [20, 21]. To support such design, an ideal simulation environment would enable the designer to experiment with a variety of formalisms and models expressed within them. This requires first, a simulation environment that can accommodate new formalisms, and second, a means of embedding and integrating formalisms into it.

We propose principles for unifying the various formalisms within a systems theoretic framework whose implementation is supported by the object-oriented paradigm. In this approach, models can be developed as instances of formalism-based classes of dynamic systems. Moreover, such an approach facilitates combining formalisms so that multiformalism models can be constructed. We present an outline of this approach before describing it in detail.

There are three basic formalisms for dynamic systems: differential equations, discrete time and discrete event system specifications (DESS, DTSS and DEVS, respectively) [22]. We consider the following ways to develop new formalisms based on these fundamental ones:

1. Specialisation: restrict a formalism so that it encompasses a narrower class of systems.
2. Generalisation: expand a formalism to encompass a more inclusive class of systems.
3. Multiformalism combination: combine formalisms together into a new formalism.

These three kinds of processes arise in more specific ways:

1. *Embedding*: interpret a new (relative to dynamic systems) formalism as a specialisation of an existing one. For example, we can formulate Petri nets as special kinds of DTSS or DEVS. This enables the new formalism to be employed within the simulation environment in the same manner that it would be employed outside it.
2. *Closure under coupling*: define a means of coupling systems expressed in a multiformalism combination and generalise the constituent formalisms so that the result is closed under coupling. Praehofer [23] has provided a very useful example of this possibility. He defined a scheme appropriate for coupling together components from the basic DESS and DEVS formalisms, and introduced the DEV&DESS formalism to represent the closure under coupling of the multiformalism.

The operations just outlined offer a powerful means of expanding the expressive capabilities of modelling and simulation environments within the CAST framework. In the sequel we review the background of systems and simulation theory needed to develop tools to support such formalism extensions. We then proceed to discuss our approach to developing such tools based on object-oriented implementation concepts.

2. Review of Systems Theory

Although a generally accepted definition of 'system' does not exist, we adopt the following definition: a *dynamical system* is any formal construct which provides general modelling concepts for various kinds of disciplines [2,22,24]. We distinguish such a mathematical object from any reality that it may represent, using the term *real system* for the latter.

A real system can be represented at varying levels of abstraction. According to the abstraction level, the system manifests itself in different ways and we use different terms to speak about it. By *system behaviour* we denote the way the system appears on its boundary, i.e. how it reacts to inputs by producing outputs. The interior of a system is described by the *system state* and the *system dynamic*. The system state represents the condition the system is in at a particular time and the system dynamic governs the way this state changes over time. When the state is represented by one or more variables we speak of *state variables*. The dynamic of the system can be determined by a state transition function which depends on the input and the state itself. How the state and current input appear as output is determined by the *output function*.

When we identify several elements corresponding to parts of the real world we speak of *system elements* and the state of each of the elements is represented by the system element state. The interdependencies of these elements contributing to the system dynamic is called the *system structure*. When we go further and represent parts of the

system by systems themselves and their interdependencies by coupling these parts, we speak of *system components* and *system couplings*. Such a system is called *multicomponent system* or *coupled system*.

2.1. Levels of System Description

Zeigler [22] defines a hierarchy of levels of system descriptions stratified according to their level of abstraction.

Each level introduces more concreteness into the description of the internal structure of a system. In the following we review this hierarchy, starting from the most abstract level and proceeding to the most concrete level of description.

Level 0. Observation Frame $O = (T,X,Y)$. Using an observation frame O we just define the system boundary. The set X is the input interface in the form of a set of inputs and Y is the output interface in form of a set of outputs. The set T is called the time base. It is used to order events and represents the observation times, i.e. points in time when we are able to define system behaviour and system dynamic. Usual time bases are the real numbers (the continuous time base) or the integers (a discrete time base).

Level 1: Relation Observation IORO $= (T,X,\Omega,Y,R)$. The relation observation IORO defines the behaviour of the system using a relation R of possible input values at particular times and their possible output values. Hence there is not an unequivocal mapping of input values to outputs but for a particular input there can be several different outputs.

Level 2: Function Observation IOFO $= (T,X,\Omega,Y,F)$. This level is equal to level 1 with the only difference being that the relation R is partitioned into a set F of functions f and each function defines a unique output response for a particular input segment. Hence, when we have knowledge of the function f, we have knowledge of the 'initial state' to get a unique output response.

Level 3: An I/O System (or Dynamical System) IOS $= (T,X,\Omega,Q,Y,\delta,\lambda)$. Whereas in level 0 to 2 it only was possible to represent the system boundary and the system behaviour, in this level we are able now to model the interior of the system. For that we use the set of states Q. The global state transition function δ realises the dynamic of the system. The output function λ is used to model how the current state manifests at the output interface. Level 3 incorporates our most important modelling concept for simulation modelling and therefore we also denote an I/O system dynamical system.

Level 4: Structured I/O System SIOS $= (T,X,\Omega,Q,Y,\delta,\lambda)$. A structured I/O system is an I/O system where the set of inputs, outputs and states are multivariable sets, i.e. we are able to identify several input, output and state variables.

Level 5: Coupled System CS $= (T,X,Y,Components,Coupling)$. In a coupled system the interior of the system is represented by identifying several component systems and the couplings between them. The coupling defines how the coupled system inputs and the component system outputs are mapped into component system inputs and coupled system outputs. The state of the coupled system is built up by the states of all component systems. The system dynamic is built up by the dynamic of the components and the coupling scheme. To qualify as a system, a coupled system has to have a description at level 3, i.e. there must be a way to associate with it a dynamical system description (see [22] for details on how this is done).

3. System Formalisms

Systems theory affords an integrative view of the diversity of formalisms. Indeed, it regards *formalism* as a modelling language used to define (actually select) a subset of systems. Once the subset is identified, a formalism need express only those features that distinguish a particular system from others in the same subset [25]. In this sense, a system formalism can be regarded as a short-hand means of system specification.

Basic system formalisms are *differential equation* system specifications (DESS), discrete time system specifications (DTSS), and discrete event system specifications (DEVS). The levels of system descriptions and the system formalisms build a crossproduct relation where every combination of system formalism and system level represents a possible modelling concept. Table 1 depicts the constraints imposed by the system formalisms at level 3, the input/output system level. The formalisms impose appropriate constraints on the time base, input, output and state sets, and input, output and state trajectories. Such constraints circumscribe the systems that can be members of the subset specified by a formalism.

Table 1. Constraints imposed by system formalisms at the input/output system level.

	Differential equations	Discrete time systems	Discrete event systems
Time base T	Continuous reals	Discrete integers	Continuous reals
Basic sets X,Y,Q	Real vector spaces	Arbitrary	Arbitrary
Input segments	Piecewise constant	Sequences	Discrete event segments
State trajectories	Continuous	Sequences	Piecewise constant
Output trajectories	Continuous	Sequences	Piecewise constant

Table 2. Constraints imposed by system formalisms at the coupled system level.

	Differential equations	Discrete time systems	Discrete event systems
Time base T	Continuous reals	Discrete integers	Continuous reals
Basic sets X, Y, Q	Real vector spaces	Arbitrary	Arbitrary
Input segments	Piecewise constant	Sequences	Discrete event segments
Components	Differential equation systems	Discrete time systems	Discrete event systems
Couplings	No algebraic cycles	No algebraic cycles	No direct feedbacks

4. Modular, Hierarchical Model Construction

Level 5 of the above hierarchy provides a powerful means of constructing models. This is to build them from simpler component models using coupled system specifications [22,26]. However, it is not always the case that coupling component models results in a well-defined system (i.e. one having a level 3 description). This motivates us to consider subsets of systems that support convenient coupled system construction and to further elaborate existing definitions [22] for this purpose.

We say that a subset of systems *supports coupling* if any coupled system specification, whose components are all members of the subset, has a description at level 3. A subset of systems is *closed under coupling* if it supports coupling and the coupled system, described at level 3, is a member of that subset as well.

In explanation, a subset of systems that supports coupling has the property that coupled models built from its members can be themselves employed as components in larger coupled systems. This property makes it possible to construct *hierarchical systems*.

A subset of systems that is closed under coupling has the useful property that all hierarchical models constructed from it are members of the subset as well. This makes it possible to design engines, such as abstract simulators [25], that uniformly handle such hierarchical models.

Table 2 summarises the constraints imposed by system formalisms at the coupled system level of description. Note that in addition to constraints on the elements introduced at level 3, there are constraints on the components as well as on the couplings. These constraints are necessary to ensure that the formalisms are closed under coupling. (If the components and couplings in a coupled model adhere to the restrictions imposed by the formalism, then the coupled model will itself fall within the subset delineated by the formalism.) Since the subsets are characterised by the formalism, the only way to prove that such closure holds for a formalism is to demonstrate that an arbitrary coupled model adhering to the constraints is 'equivalent' to a system specified in the formalism (see [22,25] for details).

5. SES Representation for System Specification Formalisms

The system entity structure (SES) [27] is a means to represent a system to be modelled within a certain choice of system boundary. It is a tree-like graph that encompasses the boundaries, decompositions and taxonomic relationships that have been perceived for the system being modelled. In the graph we distinguish three kinds of nodes: entity, aspect and specialisation. An entity signifies a conceptual part of reality. An aspect names a possible decomposition of an entity. A specialisation node facilitates the representation of variants of an entity and has similar semantics to the concept of specialisation in the object programming paradigm. Each of these nodes can have attached variables. Typically, the SES is employed to specify families of design or simulation models, generated by pruning a master SES, for a given application domain. Here, we employ SES concepts to provide the knowledge

representation structure needed to manage system specification formalisms.

An SES-based knowledge-representation scheme to facilitate management of system specification formalisms is shown in Fig. 1.

The root entity of this SES, called the canonical SES, represents a system specification S – either an atomic or a multicomponent one. The constituents of S are sets and functions. Sets can be classified into Input, Output, Time and State as well as sets of component names or other, modeller-defined objects (e.g. an initial state in a finite state machine specification). Each set can be characterised by the type of its elements, e.g. reals, integers, etc. In addition, by selecting the system-specification variant from the Set entity, we can include a set of system specifications as elements of the system specification S. This recursive representation affords the construction of formalisms for multicomponent systems. (Note that we must relax the strict hierarchy SES axiom as in Cho [28] in order to facilitate such a process.)

The functions are: Transition, Output, and Segment, I/O Translation, or Other (i.e. modeller-defined functions, for example, a rate of change, or time advance function). Each function has an attribute type which characterises its properties, e.g. step, piecewise continuous, piecewise differentiable, etc.

In a system design process, the canonical SES of system specifications is used in conjunction with an SES for an application domain. Pruning the application SES generates a composition tree for the system model specification. The model composition tree is a tree whose leaf nodes are system specifications. These are atomic components which are coupled in a hierarchical manner. In the next section, we show how both atomic and coupled level specification can be constructed based on the composition tree and canonical SES representations.

As an example, Fig. 2a depicts a composition tree of a two component system. Assume, that the two subsystems S_1 and S_2 are connected in series as shown in Fig. 2b.

The resultant is the system S whose formal specification should now be derived. Let us illustrate this process at both atomic and the coupled system level.

5.1. Atomic System Specification

To generate a formal specification for the atomic components S_1 and S_2, design constraints and requirements as well as physical characteristics of the model's counterpart (i.e. real system) are analysed. This is done in order to determine appropriate types of sets and functions needed to characterise dynamic behaviour of the components under consideration.

Assume that both subsystems exhibit an inherently discrete behaviour. The modeller may prune the canonical SES so that the DEVS formalism is selected for formalism-type. Since DEVS is formalism that will be known to the system, its constituent slots will be automatically retrieved leaving only their values to be supplied by the modeller. Such a selection might be presented in the form of a frame as follows.

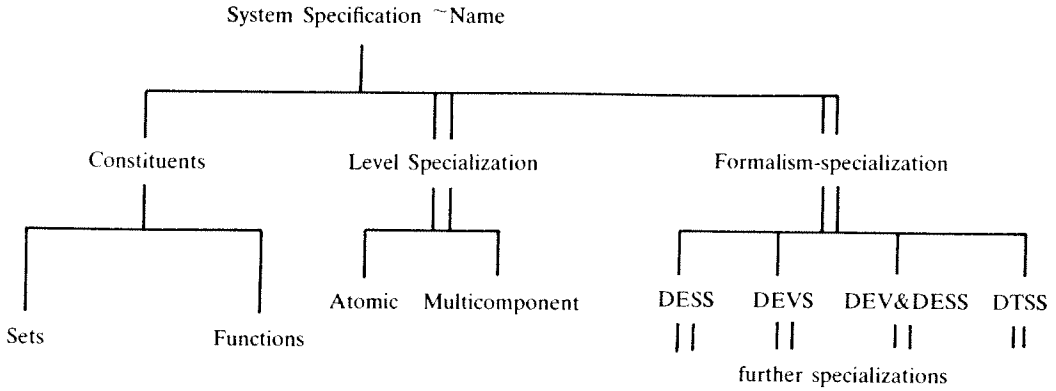
Pruned Canonical SES for an Atomic Specification (DEVS)

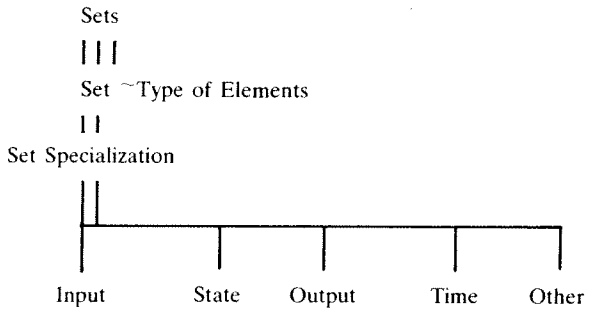
FRAME S_1
System Specification Type: Atomic
Constituents: Sets, Functions
Sets:
Time: Type of Elements: Reals
Input X: Type of Elements: Discrete Events
Output Y: Type of Elements: Reals
State S: Type of Elements: Nonnegative Integers
System Specification: None
Component: None
Other: None
Functions
Transition:
Internal: δ_ϕ: $S \rightarrow S$
External: δ_{ex}: $Q \times X \rightarrow S$
where Q is given by:
$Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$
Output: λ: $S \rightarrow Y$
Other: Time advance function:
ta: $S \rightarrow R_{0,\infty}$
Input-to-Output Translation: None
Segment: None

Choosing a DEVS atomic model for component S_2, a frame S_2 is developed similar to that of frame S_1. We now proceed to illustrate how to construct a coupled level system specification.

5.2. Coupled System Specification

The specification of the system S can be obtained by coupling the specifications of its components. Since the DEVS formalism is closed under coupling, the modeller may prune the canonical SES to select the DEVS formalism for the multicomponent level specialisation.





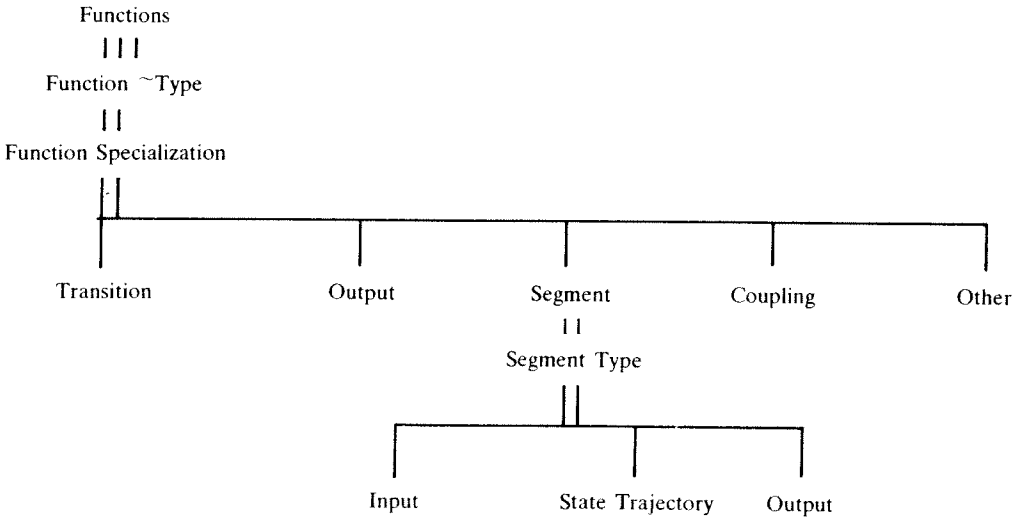


Fig. 1. Canonical SES of system specification elements.

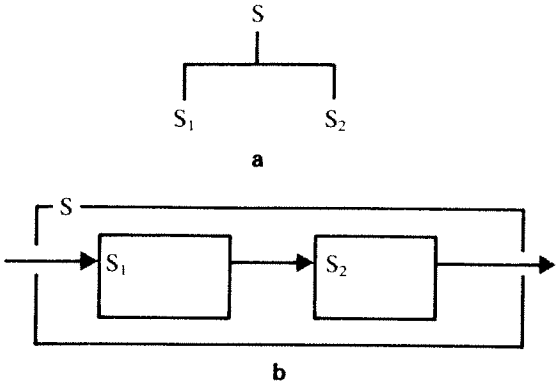


Fig. 2. **a** A simple composition tree. **b** System S as a coupling of subsystems S_1 and S_2.

Pruned Canonical SES for a Multicomponent Specification (DEVS)

FRAME S
System Specification Type: Multicomponent Discrete Event Network
Constituents: Sets, Functions
Sets:
Component:
D – component names S_1, S_2
I – influences: S_1 : {}, S_2: $\{S_1\}$
System Specifications:
Frame S_1
Frame S_2
Functions:
Coupling (Input-to-Output Translation):
Z_{12}: $Y_1 \rightarrow X_2$ defined as the identity mapping
Other:
Select: $2^D \rightarrow D$, defined by linear order (1,2)

Note that in environments such as DEVS-Scheme [20] and STIMS [29] the slots in the above frames can be filled in a manner that is user-oriented yet very close to their 'pure' mathematical forms.

Note also that constraints such as those given in Table 2 must be added to the SES to ensure the compatibility of component formalisms with the coupled system formalism at the next higher level.

6. Object-Oriented Implementation of the Canonical SES

We have now shown how the canonical SES can structure modelling formalisms. Thus we are ready to show how the implementation of the multiformalism modelling and design environment can be supported by the object-oriented programming paradigm. We show how the SES-based knowledge representation scheme can be mapped to CLOS [30] employing its class definitions, multiple inheritance, and methods of different method roles.

In a realisation of the multiformalism modelling and design framework outlined above, we must represent the following:

- System formalisms
- Constituents of a formalism (sets and functions)
- Formalism specialisation hierarchy
- Constraints and restrictions defined for the formalism and its constituents
- Dynamic characteristics of the formalisms (its simulation algorithm in a multiformalism framework)
- Operations defined for the formalisms.

6.1. System Formalisms

Mapping of system formalisms and models to an object-oriented environment is straightforward. The formalisms can be represented by class definitions, the models of the different formalisms are realised by instances of the particular formalism classes.

6.2. Constituents of the Formalism

The constituents of the models are defined by slot (also called instance variable) definitions. Operations on these slots in the different formalism classes can be unified by the generic function concept of CLOS. Here the polymorphism of methods in the object-oriented paradigm shows its usefulness. Operations can have different realisations in different formalisms. For example, computing a state transition can be represented by one generic function having a different method implementation in each formalism class. This has the advantage that the model specification is always done through a well defined interface. Also the actual implementation of the system formalisms and their constituents is hidden from the user and may be subject to change without affecting the user interface.

6.3. Formalism Specialisation Hierarchy

To represent the specialisation relationship of the different formalisms, the object-oriented paradigm

offers the concept of class hierarchy with multiple inheritance. However, due to the diversity of the formalisms and their great variety of interrelations, such a representation proves to be difficult. Mittelmann and Praehofer [31] proposed a knowledge representation scheme where the constituents of system formalisms are implemented by abstract classes whereas the actual formalisms are defined through dynamic class definitions multiply-inheriting from the appropriate constituent classes. They also provide an algorithm to set up an inheritance hierarchy which minimises the duplication of slots and methods in a complex specialisation hierarchy, an objective of any object-oriented implementation. This approach (see Appendix) works by listing all the classes together with all their slots and methods. Then from this set of all slots and methods, common slots and methods of different classes are identified and they are taken out to be implemented in common abstract superclasses only once and then inherited to all the classes needing the slot or method. This algorithmic approach has been employed in the design of the formalism class hierarchy of the STIMS modelling and simulation environment [29] with great success. STIMS is an environment implementing DEV&DESS-based multiformalism modelling and simulation. Its formalism class system constitutes of about 40 classes organised in a complex inheritance hierarchy. With the algorithmic approach the handling of the class inheritance has become manageable.

Figure 3 shows part of an example formalism class hierarchy implementing several different coupled system formalisms. Actual system formalism classes are the classes at the bottom of the hierarchy. The inner classes are abstract classes used to hold common slots and methods of subclasses. The real classes can be created by dynamic class definitions as outlined in [31], i.e. not all combinations of abstract classes defining coupling and type specific features have to be created in advance, but can be created on demand.

The class hierarchy developed in this way can be reflected in the structure of the canonical SES, so that pruning the latter results in a set of superclasses that combine to compose a dynamic class.

6.4. Defining Formalism-Specific Constraints

Methods can also be used to implement the constraints defined on the constituents of system formalisms. This will be outlined in the following.

Meyer [32] discusses the use of assertions that check the situation before and after method invocations. The precondition expresses the assumptions that are required to hold whenever the method is called. The postcondition describes the properties that must hold after the method has been executed.

In CLOS, so-called "before" and "after" methods can be employed to specify pre- and postconditions. In particular, before methods can express preconditions to check the fulfilment of the constraints on formalism constituents. Since before methods are specified independently from the primary method, the specifications of the preconditions can be distributed in the class hierarchy and implemented in the most appropriate abstract class. Also since before methods are not shadowed (in other words are unconditionally inherited) and are called in

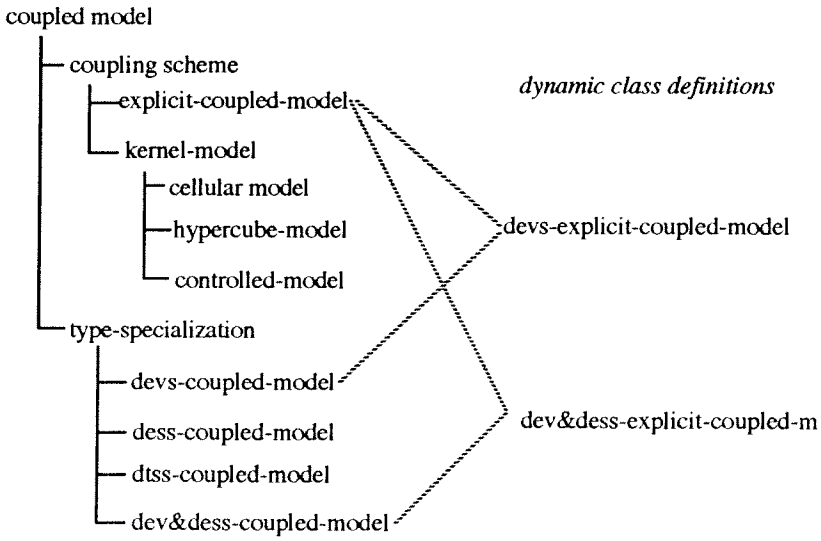


Fig. 3. Example system formalism implementation employing dynamic class definitions.

most-specific-first order, it is guaranteed that all constraints which apply to a particular formalism are tested before the primary method is called.

For example, the function to specify a coupling in a multiformalism coupled model consists of a primary method which actually stores the coupling. Of course, this primary method is specialised on the class implementing the coupling, e.g. in class *explicit-coupled-model* in the formalism class hierarchy above. However, formalism-specific constraints apply for couplings. For example, in a multiformalism coupled model, a coupling from a continuous port to a discrete port is not allowed. Therefore, a before method specialised to the class *dev&dess-coupled-model* checks for this constraint violation. As a second example, recall that direct feedback loops are not allowed around discrete event models. This constraint is tested in a before method specialised on class *devs-coupled-model* and is also inherited by coupled *dev&dess-coupled-model*. Hence, before calling the primary method for adding a coupling to a model of class *dev&dess-explicit/coupled-model*, the two just-mentioned before methods would be called to check their respective constraint preconditions. Only when these checks are passed, would the primary method add the coupling.

6.5. Simulation Algorithms

The separation of the knowledge base from the inference engine is a commonly accepted practice in AI programming. In systems modelling and simulation, the separation of the model specification and the simulation engine is accomplished by the abstract simulator concepts for DEVS models introduced in [25] and its multiformalism extensions introduced in [23] and [33]. The abstract simulator for modular hierarchical models is a composition of objects reflecting the structure of the model. Coordinators are associated with the coupled models, simulators are associated with the atomic models. In [33] special coordinators and simulators are introduced to do the numerical integration of continuous states. The simulation task is carried out by passing messages between these objects.

Object-oriented implementation of the abstraction simulator concepts is straightforward. The various types of coordinators and simulators associated with different model formalisms, are implemented by class definitions. In CLOS, a generic function with method implementations specialised on the formalism class of the model can be used to create the appropriate simulator or coordinator object for a particular component model.

Simulation execution by message passing is implemented by generic function calls. The polymorphism of generic functions efficiently supports the simulation of multiformalism-coupled models. In simulation of a multiformalism-coupled model, the coordinator of the multiformalism-coupled model sends and receives messages to and from its subordinate simulators and coordinators. Due to the uniformity of interface, it does not have to know the types of the subordinates in this interchange. Of course, this requires that the subordinate simulator and coordinators adhere to the same generic function interface, i.e. they have to send and receive the same generic functions. Of course the reaction of the subordinates to the different messages received will depend on the type of the simulator or coordinator as expressed in the different method implementations in the simulator and coordinator classes. This scheme is a typical example of task delegation often employed in object-oriented programming. The sending coordinator does not want to know what the subordinates actually do with the messages, only that they will react appropriately.

6.6. Integration of New Formalisms

We now outline how an object oriented implementation designed according to the above principles supports the integration of a new formalism into an existing multiformalism framework. To accomplish this, we have to implement the new formalism with all its constituents and operations as a class and integrate this class into the existing formalism class hierarchy. In addition, we have to provide the simulation concepts to allow models of the new formalism as components in a multiformalism simulation model.

The first task can be accomplished when the inheritance hierarchy is designed employing the approach of [31] (and reviewed in Appendix A). All slots and methods of the new formalism class have to be listed. Discovering its common slots and methods, the new formalism will find its appropriate place in the inheritance hierarchy. Its place will reflect its relations to the other formalisms. The difficulty of this task is greatly reduced when the to-be-integrated formalism can be embedded into an existing atomic model formalism. This means that a mapping is provided to translate the constituents of the new formalism into the constituents of the existing one. In this case, many of the new slots employed in the existing atomic model formalism may be inherited by the new formalism and only those which must be specialised to represent the embedding need be redefined for the new formalism.

To accomplish the second task, special simulation classes have to be implemented. These classes have to realise the special behaviour of the formalism models in a multiformalism-coupled model. They have to be able to react to the same messages as all other component simulators or coordinators. Formally, this requires that the expanded set of systems (specified by the existing and newly added formalism) be closed under coupling. Once again, the task is simplified for the case where a formalism is embedded in an existing atomic model formalism. In this case, only the simulator for atomic models of the new class need be designed – the coordinator for the superior multiformalism coupled model need not be changed. An example of such proof of closure and the associated coordinator design is that developed for the DEV&DESS formalism and is discussed in detail elsewhere [23,29].

7. Conclusions

This paper has demonstrated the utility of adopting the formal systems approach to integrating classes of systems formalisms within a modelling and design environment. Especially when applied to intelligent system design, the number of formalisms being proposed by researchers is growing tremendously. Our approach is intended to make such formalisms attractive to researchers, in the first instance, and eventually to designers of an integrated environment. An alternative approach, the multimodelling framework suggested by Fishwick [34], is to provide an environment that allows the integration of different formalisms at different levels of abstraction. Such an environment, however, attempts to unify the various formalisms and support multiformalism modelling, viz. the ability to employ different formalisms within different components of the same model. Although we have not provided a working prototype of such an environment in this paper, we have cited some that exist, and we have demonstrated that the conceptual principles and the object-oriented programming environments to implement them are at hand.

References

1. Bossel H. Systemdynamik. Vieweg, Sohn, Braunschweig, 1987
2. Pichler F. Mathematische Systemtheorie. Walter de Gruyter, Berlin, 1975
3. Pichler F, Schwartzel H. CAST: Computer-aided system theory. Springer Verlag, Berlin, 1990
4. Klir GJ. Architecture of system problem solving. Plenum Press, New York, 1985
5. Mesarovic MD, Macko D, Takahara Y. Theory of hierarchical, multilevel systems. Academic Press, New York, 1970
6. Pichler F, Schwartzel H. CAST (Computer aided system theory) methods in modelling, Springer-Verlag, New York, 1992
7. Ho Y-C. Special issue on discrete event dynamic systems. Proc IEEE 1989: 77(1)
8. Lee CC. Fuzzy logic in control systems: fuzzy logic controller – Part I. IEEE Trans Syst Man Cybern 1990; 22
9. Kosko B. Neural networks and fuzzy systems: a dynamical systems approach to machine intelligence. Prentice-Hall, 1992
10. Miachalewicz Z. Genetic algorithm + data structure = evolution programming. Academic Press, New York, 1992
11. Fishwick PA, Zeigler BP. Qualitative physics: towards automated systems problem solving. J Exp Theor 1991; 219–246
12. Hammond KJ. Case-based planning. Academic Press, 1989
13. Tsatsoulis C, Kashyap RL. A case-based system for process planning. Robot Comput Integr Manuf 1988; 4(3/4), 557–570
14. Shafer G, Pearl J. Uncertain reasoning. Morgan Kaufmann Publishers, San Mateo, CA, 1990
15. Genesereth MR, Nilsson NL. Logical foundation of artificial intelligence. Morgan Kaufmann Publishers, 1987
16. Antsaklis PJ, Passino KM. Introduction to intelligent control systems with high degrees of autonomy. In: Antsaklis PJ, Passino KM (eds). An introduction for intelligent and autonomous control. Kluwer Academic Publishers, Norwell, MA, 1992
17. Saridis GN. Analytic formulation of the principle of increasing precision with decreasing intelligence for intelligent machines. Automatica 1989; 25(3), 461–467
18. Albus JS. A reference model architecture for intelligent systems design. In: Antsaklis PJ, Passino KM (eds). An introduction to intelligent and autonomous control. Kluwer Academic Publishers, Norwell, MA, 1992
19. Zeigler BP, Chi SD. Model-based architecture concepts for autonomous systems design and simulation. In: Antsaklis PJ, Passino KM (eds), An introduction to intelligent and autonomous control. Kluwer Academic Publishings, Norwell, MA
20. Zeigler BP. Object-oriented simulation with hierarchical, modular models. Academic Press, New York, 1990
21. Zeigler BP, Cellier FE, Rozenblit JW. Design of a simulation environment for laboratory management by robot organizations. Intell Robot Syst 1989; 1: 299–309
22. Zeigler BP. Theory of modelling and simulation. John Wiley, 1976
23. Praehofer H. Systems theoretic foundations for combined discrete continuous system simulation. PhD thesis, Department of Systems Theory, University Linz, Austria, 1991
24. Wymore AW. Systems engineering methodology for interdisciplinary teams. John Wiley, New York, 1976

25. Zeigler. Multifacetted modelling and discrete event simulation. Academic Press, New York, 1984
26. Bossel H. Systemdynamik. Vieweg, Sohn, Braunschweig, 1987
27. Rozeblit JW, Zeigler BP. Design and modelling concepts, encyclopedia of robotics. John Wiley, 1988, pp 308–322
28. Cho T. A hierarchical, modular simulation environment for flexible manufacturing, Doctoral Dissertation, University of Arizona, Tucson, 1993
29. Praehofer H, Auernig F, Reisinger G. An environment for DEVS-based modelling in common Lisp/CLOSS. J. Discrete Event Dyn Syst Theory Applic (to appear)
30. Keene SE. Object-oriented programming in common LISP. Addison Wesley, 1989
31. Mittelmann R, Praehofer H. Design of an object oriented kernel system for computer aided systems theory and systems theory instrumented modelling and simulation. In: Computer aided systems theory – EUROCAST T89, eds. Pichler F, Moreno-Diaz R. Lecture Notes in Computer Science, Springer-Verlag, 1990, pp 76–85
32. Meyer B. Object oriented software construction. Prentice Hall, New York 1988
33. Reisinger G. Simulation of combined discrete-continuous, modular, hierarchical models in common Lisp/CLOS, Master thesis, Johannes Kepler University, Linz, Austria, 1992

Appendix A. Organisation of Inheritance Hierarchies

In the following we will outline an approach to set up an inheritance hierarchy and to avoid code redundancies introduced in [31]. The approach solves the following problem: a set of classes and all their features are given. Find the optimal inheritance hierarchy so that common features of different classes are only implemented once. Definitions:

- Real classes are classes from where instances should be created.
- Abstract classes are classes from where no instances are created. These classes are intended to hold common features of real classes.

Features are slots and methods defined for a class. The algorithm works as follows in six steps:

1. Determine the set $rC = \{r_1, r_2, \ldots, r_m\}$ of real classes.
2. For each real class r_i determine the set $F_{ri} = \{f1, \ldots, fn\}$ of features $f_1, f_2, \ldots, f_n$ for the class.
3. Compute the transitive closure K of $F_{r1}, \ldots, F_{rm}$ with respect to the intersection operation $\cap$, i.e. with all F_{ri} pairwise and groupwise compute the intersection until no new sets are gained.

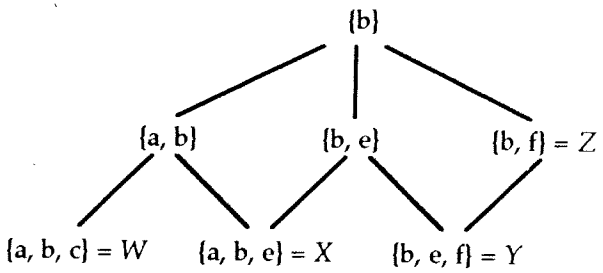


Fig. A.1. Graph of subset relation.

4. Each element of K must be implemented in one class.
5. The inheritance hierarchy is determined by the subset relation.
6. Draw a diagram of the inheritance hierarchy and remove all features which already are inherited from a superclass.

In the following, an example is shown to clarify the idea.

1. W, X, Y and Z denote our real classes.
2. These classes own the following features:

$W : F_W = \{a,b,c\}$ $X : F_X = \{a,b,e\}$ $Y : F_Y = \{b,e,f\}$ $Z : F_Z = \{b,f\}$

3. The transitive closure $K : F_W \cap F_X = \{a,b\}$, $F_W \cap F_Y = \{b\}$, $F_W \cap F_Z := \{b\}$, $F_X \cap F_Y := \{b,e\}$, $F_X \cap F_Z := \{b\}$, $F_Y \cap F_Z := \{b,f\}$.

$K := \{\{a,b,c\},\{a,b,e\},\{b,e,f\},\{ab\},\{b\},\{be\},\{b,f\}\}$

4. Besides W, X, Y and Z, we need three more classes to implement the subsets $\{ab\},\{b\},\{be\}$.
5. Figure A.1 shows a diagram of the class hierarchy where inheritance relation is determined by the subset relation.
6. Remove all features which are already inherited from any superclass to obtain the final class hierarchy (Fig. A.2). The remaining features are the features which have to be implemented in the class.

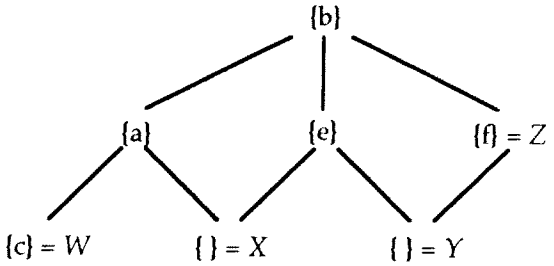


Fig. A.2. Optimal inheritance hierarchy.