Softw Syst Model
DOI 10.1007/s10270-007-0079-3

REGULAR PAPER

Reusing semi-specified behavior models in systems analysis and design

Iris Reinhartz-Berger · Dov Dori · Shmuel Katz

Received: 3 January 2006 / Revised: 13 May 2007 / Accepted: 13 December 2007
© Springer-Verlag 2008

Abstract As the structural and behavioral complexity of systems has increased, so has interest in reusing modules in early development phases. Developing reusable modules and then weaving them into specific systems has been addressed by many approaches, including plug-and-play software component technologies, aspect-oriented techniques, design patterns, superimposition, and product line techniques. Most of these ideas are expressed in an object-oriented framework, so they reuse behaviors after dividing them into methods that are owned by classes. In this paper, we present a crosscutting reuse approach that applies object-process methodology (OPM). OPM, which unifies system structure and behavior in a single view, supports the notion of a process class that does not belong to and is not encapsulated in an object class, but rather stands alone, capable of getting input objects and producing output objects. The approach features the ability to specify modules generically and concretize them in the target application. This is done in a three-step process: designing generic and target modules, weaving them into the system under development, and refining the combined specification in a way that enables the individual modules to be modified after their reuse. Rules for specifying and combining modules are defined and exemplified, showing the flexibility and benefits of this approach.

Communicated by Dr. Kevin Lano.

I. Reinhartz-Berger (✉)
University of Haifa, Carmel Mountain, 31905 Haifa, Israel
e-mail: iris@mis.haifa.ac.il

D. Dori · S. Katz
Technion-Israel Institute of Technology,
Technion City, 32000 Haifa, Israel
e-mail: dori@ie.technion.ac.il

S. Katz
e-mail: katz@cs.technion.ac.il

Keywords Software reuse · Aspect-oriented software engineering · Aspect-oriented modeling · Object-Process Methodology · Modularity

1 Introduction

The last few decades have witnessed increasing interest in software reuse, i.e., the use of existing software artifacts or knowledge to create new software [20]. Software reuse aims at improving software quality and productivity by integrating existing modules, such as commercial off-the-shelf (COTS) products or tested modules from other projects. Early software reuse emphasized combining reusable source code modules to produce application software [39]. The object-oriented paradigm has highlighted the importance of reusability as part of the entire software system development process by using classes, packages (modules), and the inheritance mechanism as primary linguistic vehicles for reuse [8]. The current definition of software reuse encompasses the variety of resources that are generated and used during the development process, including requirements, architecture, design, implementation, and documentation. In this paper, we focus on reusing analysis and design models.

Software engineering approaches refer to reuse in various ways. Plug-and-play software component technologies incorporate existing complete, stand-alone modules, such as packages or classes, into new applications via the modules' interfaces [6,38]. Design patterns describe key aspects of successful solutions to design problems along with the benefits and tradeoffs related to using those solutions [22,36,56]. Reuse of design patterns is usually achieved through parameterization and binding capabilities. Aspect-oriented and superimposition methods separate the system being developed into several different aspects or concerns that

might cut across the system structure [2,5,9,10,31]. These aspects are woven and integrated into the complete system by copying, inheriting, or wrapping them. Product line techniques, such as [21], explicitly capture the commonality and variability in the family of systems that constitutes the product line. These techniques reuse the generic product line artifacts by refining them in individual applications.

Most of these approaches apply object-oriented techniques. While object-orientation has many advantages, such as standardization, encapsulation of object data and implementation, and easier maintainability, it distributes system behavior and functionality among the different structures (objects) of the system. Hence, reusing behavioral (process-oriented) modules in the object-oriented approach requires special, often unnatural and complicated techniques for incorporating reused behaviors into existing systems. In this work, we suggest Object-Process Methodology (OPM) as the basis for a general-purpose approach, which facilitates reuse of generic modules that combine structure and behavior in a unified, intertwined way. This approach can be used to augment target designs. OPM [14] extends the object-oriented paradigm with a new entity, called *process class* (or process for short), which is a pattern of transformation (consumption, generation, or change) that objects (possibly from different object classes) undergo. The system's structure, behavior, and function are unified into a single, coherent view, enabling reuse that is based on that view. The proposed reuse approach entails the design of generic, partially specified ("half baked") modules and their integration with the system under construction. After the initial integration of the modules, the system is further enhanced ("fully baked") and optimized in a way that best suits the task at hand while maintaining the original core function of the individual modules.

The paper is organized as follows. Section 2 reviews existing techniques for reusing specifications and designs. It argues that these techniques become complicated and inadequate when the modules are process-oriented. Section 3 briefly describes OPM, while Sect. 4 specifies rules for weaving and integrating modules in OPM and discusses the semantics of the resulting woven modules. Section 5 demonstrates the approach on a case study of a Web-based accelerated search system and discusses our findings. Finally, Sect. 6 summarizes the contributions of this work and relates to future work.

2 Reuse of design modules in modeling techniques

Current object-oriented modeling techniques and languages, notably UML [41], emphasize the importance of reuse during the development process and facilitate it through classes, packages, and the inheritance mechanism [33,37]. From a reuse perspective, once the classes and packages have been modeled, they are usually treated as closed, black boxes with interfaces, through which other parts of the module or other modules can communicate. While this approach improves maintainability of large systems, it hinders reusing generic modules with behavior that crosscuts the structure of the application under development. Mezini and Lieberherr [38] claim that object-oriented programs are more difficult to maintain and reuse because their functionality is spread over several classes, making it difficult to get the "big picture." They suggest designing modules that facilitate the construction of complex systems in a manner that supports the evolutionary nature of both structure and behavior.

To respond to this challenge, aspect-oriented programming (AOP) [31,54] modularizes the features for a particular concern that cuts across multiple classes and enables these features to be woven, i.e., incorporated and integrated, into the system model at the level of programming languages. Superimposition language constructs [5,7,9,29] similarly extend the functionality of process-oriented systems, again cutting across the software architecture of process hierarchy. These techniques allow the imposition of predefined, yet configurable, types of functionality with reusable modules.

Several attempts have been made to extend the aspect notion from programming to "early aspects" [19]. These attempts deal with crosscutting concerns in the early life cycle phases, including requirements analysis, architecture design, and detailed design stages. The "early aspects" approaches handle different activities, including aspect identification, aspect representation, weaving guidelines definition, and weaving application. They differ in the phases they mainly support, in their declared goals, in the supported crosscutting elements, and in their specification means. Most of them use UML as their modeling language and employ different sets of stereotypes to model the new aspect-oriented concepts.

Katara and Katz [28] view aspects as augmentations that map an existing design artifact into a new one with changes or additions. Their framework, which is applied to UML, has some similarities to our approach, but since they use UML, they must treat each diagram type separately, requiring special consistency treatment. They also need to indicate dependencies and interactions among aspects.

Baniassad and Clarke [4,12] have promoted the notion of a "Theme", which is a design element or a view of the system object that helps focus on only a portion of the object model that is relevant to a particular piece of functionality. The approach is divided into two parts: Theme/Doc, which enables identifying and isolating concerns in the requirements engineering phase, and Theme/UML, which aims at modeling those concerns. Within Theme/UML [11,13], UML is extended with two types of composition relations, merge and override, which involve an entire UML unit

(e.g., attribute, method, or class). This extension handles (at least for now) only class and sequence diagrams.

Kande [27] proposes a perspectival concern-space (PCS) technique for depicting concerns of multiple dimensions in an architectural view consisting of one or more models and diagrams. The PCS framework provides means for composing and decomposing different concern spaces. The goal of this approach is to develop architecture with concerns as primary dimensions. The aspect-oriented generative approach (AOGA) [32] is an architecture-centric method whose purpose is to support multi-agent system development by providing domain-specific languages, modeling notations, and code generation tools. In addition to central (knowledge) components that modularize the non-crosscutting features associated with the agent knowledge, they provide aspectual components that separate the crosscutting agent features from each other and from the knowledge components. The aspect-oriented design model (AODM) [55] introduces UML stereotypes, tagged values, and constraints that enable representing the aspect-oriented design concepts that are specified in AspectJ [3]. Aldawud et al. [1] present a UML profile and stereotypes that extend UML semantics and specify the structure of system and aspect models in terms of class diagrams, while behavior is modeled using statechart, use case, state machine, and collaboration diagrams. There are also approaches intended to treat the entire software development cycle [2,24].

Many of these and other "early aspects" approaches reuse programming-level concepts without adaptation. Hence, they fall short of considering the entire spectrum of modeling concepts not present in programming languages, such as different views of the application's structure and behavior [50].

In the context of reusable components, Catalysis [17] is a methodology for component and framework-based development that provides consistency rules across models, and mechanisms for composing these views to describe complete systems. Troll [18] suggested adding parameterization and binding capabilities to UML packages.

Design patterns [22] also tackle some reuse challenges by describing common design solutions for recurring design problems. They are typically described using a combination of natural languages, UML diagrams, and program code. To overcome the unbalanced representation of the static and dynamic aspects of design patterns in UML, the notation can be extended using UML's stereotype mechanism [36]. This increases the language vocabulary and expressiveness at the price of increasing its complexity.

Product line techniques enable reuse and concretization of generic product line artifacts in some specific domain. Software product line architecture is defined for a family of products and not for an individual application. Some product line development approaches provide a generic architecture, or reference model that depicts only the commonality of the product line, ignoring variability. Each application starts with the generic architecture and adapts it as required. Although this approach provides a better starting point in comparison to developing a system without any reuse, it fails to capture knowledge about the variability in the product family [21]. Other approaches explicitly model both commonalities and differences in the product family. Depending on the development approach used (functional or object-oriented), the product line commonalities are described in terms of common modules, classes, or components, and the product line variabilities are described in terms of optional or variant modules, classes, or components (e.g., [23] and [25]).

The methods surveyed above either reuse complete structural units, such as packages, or have to deal with augmenting the variety of diagrams in ways that cut across class boundaries, complicating the reuse process. Furthermore, most methods do not relate to subsequent phases of the system development that need to be accounted for after reusing generic modules. Complete integration often requires that certain changes be made to parts of the original module units. This implies that modules cannot be black boxes, but rather white or transparent boxes, whose contents can be accessed and modified. This kind of support is often essential for optimizing and enhancing the design of an entire system, a mission that goes beyond binding existing modules together.

Although UML has advantages when focusing on a specific (usually structural) part of the modeled system and in maintaining views of a reasonable size, we found out that Object-Process Methodology is more suitable for the purpose of our work. In most real-world systems, structure and behavior are tightly intertwined, and are therefore hard to separate or isolate for reuse purposes. OPM enables direct modeling of the coexistence of systems' structure and behavior in the same view without highlighting one at the expense of suppressing the other. This way complete behaviors that cut across system structure can be reused in a clean and clear way.

3 Object-Process Methodology basics and ontology

Object-Process Methodology (OPM) [14,15,48] is a holistic approach to the modeling, study and development of systems. It integrates the object- and process-oriented paradigms into a single frame of reference. The elements of the OPM ontology are entities (things and states) and links. A *thing* is a generalization of an *object* and a *process* – the two basic building blocks of any system expressed in OPM. Objects are things that have the potential of stable, unconditional physical or mental existence. An object might have a set of *states*, each of which describes a situation at which the object can be at some point in time. Processes express behavior and enable

the transformation of objects: generation, consumption, or change of their state (or value).

Structural and behavioral aspects of a system are expressed through links. *Structural links* express static, time-independent relations between pairs of entities. Structural links specialize into general (tagged) structural links, and four fundamental structural links. A tagged structural link is usually labeled by a textual tag that is set by the system architect to convey a meaningful relation between the two linked entities. The four most prevalent and useful OPM structural relations, which are termed fundamental structural links, are *aggregation-participation*, which denotes whole-part relations, *generalization-specialization*, which gives rise to inheritance or sub-classing relations, *exhibition-characterization*, which connects a class to its attributes and/or operations, and *classification-instantiation*, which connects object or process instances to their classes.

Procedural links connect entities to processes in order to describe the behavior of a system, i.e., how processes transform and use entities. The behavior of a system is described in three major ways, expressed by three groups of procedural links: (1) transforming links: processes can transform (generate, consume, or affect) entities, so transforming links include result, consumption and effect links; (2) enabling links: entities can enable processes without being transformed by them. This type of relation is expressed by instrument and condition links. While the semantics of an instrument link is "wait until" the condition is met, a condition link expresses a branching construct (such as if or case); and (3) triggering links: entities can trigger events that potentially invoke processes. This group includes invocation links (which enable processes to invoke other processes), exception links (which invoke processes as a result of violating time constraints), agent links (denoting humans triggering processes), and general event links.

Like object-oriented languages, OPM uses packages to manage system complexity. A *package* includes a collection of elements (i.e., objects, processes, and structural and procedural links). Furthermore, OPM provides three refinement/abstraction mechanisms for managing complexity within a package. These mechanisms enable specifying a system to any desired level of detail without losing the "big picture" and the comprehension of the system as a whole. Whenever a diagram becomes too crowded or cluttered, a new diagram that describes a portion of the original diagram (for example, a process with the related things attached to it) can be created. The most used refinement/abstraction mechanism is *in-zooming/out-zooming*, in which the inner details of a thing (usually a process) are exposed/hidden within its frame. Furthermore, the execution order of sub-processes within the in-zoomed process is defined by the location along vertical axis, which specifies the flow of time from top to bottom. Hence, sequential sub-processes are depicted one on top of the other, while parallel or independent processes are presented graphically at the same height. Note that the new diagram preserves the consistency of the model with the original, ancestor diagram, so all the diagrams that together model the system are consistent.

OPM has been used to support the modeling of common system types, including real-time systems [43], ERP [53], and Web applications [49]. Experiments have shown that OPM is more comprehensive than object-oriented, multiple-view notations in modeling the dynamic and reactive aspects of systems [44,47] due to its single view and its balanced treatment of system structure (objects) and behavior (processes). Furthermore, OPM's ontological completeness was proven according to the Bunge-Wand-Weber (BWW) evaluation framework [53]. Figure 1 summarizes the main concepts in OPM, their relations, and their graphical symbols.

As an example of an OPM model consider the *Product Handling* process depicted in Fig. 2. This process zooms into three sub-processes, which are executed sequentially. First, *Product Listing* produces the *Product Report* from the *Products* in the *Product Catalog*. Then, *Product Updating* is executed, updating the relevant *Products*. Finally, *Consistency Checking* creates a *Consistency Message* which reflects the status of the *Product Catalog*. As can be deduced from the OPM notation, the *Products* are affected only by the *Product Updating* process, while the other two processes only use the information stored for the *Products* for respectively creating *Product Report* and *Consistency Message*. Furthermore, each *Product* in the *Product Catalog* may be, at each point in time, at one of two states: *proper* or *defective*.

Reuse of closed structural or behavioral OPM models is not difficult, especially due to OPM's refinement/abstraction mechanisms that enable hiding inner structure and behavior of entities while leaving their interfaces. However, there are often recurrent tasks that are tightly interwoven with the rest of the system and cut across many of its parts. For example, adding a time recording capability to a process execution includes adding a timestamp to each relevant item, adding a function that updates these timestamps, defining triggers which activate these functions, and so on. In this paper we introduce and elaborate on an OPM-based transparent reuse approach, which enables weaving modules containing behavior patterns that cut across the system structure. The key to our approach is OPM's inherent interplay and the resulting synergism between structure and behavior.

4 Weaving OPM modules

An *OPM module* (or *module* for short) is a model of a system, a subsystem, or an aspect. A module can be structural, behavioral, or a combination of both. We distinguish between a generic module, which is only partially specified, and a

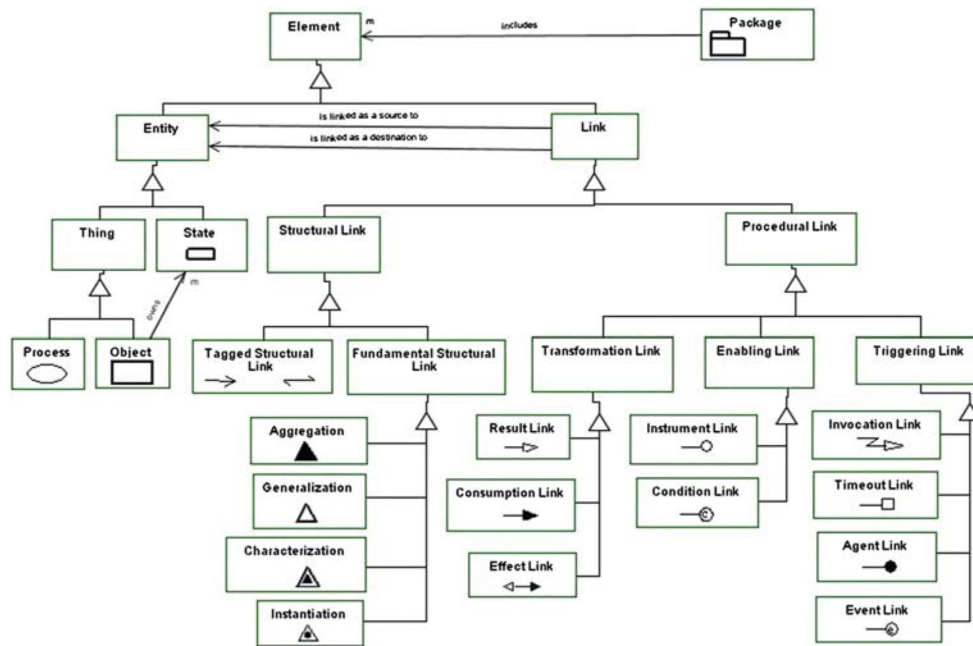

Fig. 1 OPM concepts, relations, and graphical symbols

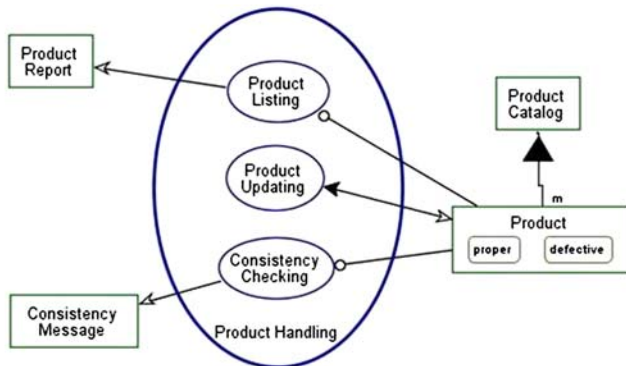

Fig. 2 An OPM model of a product handling process

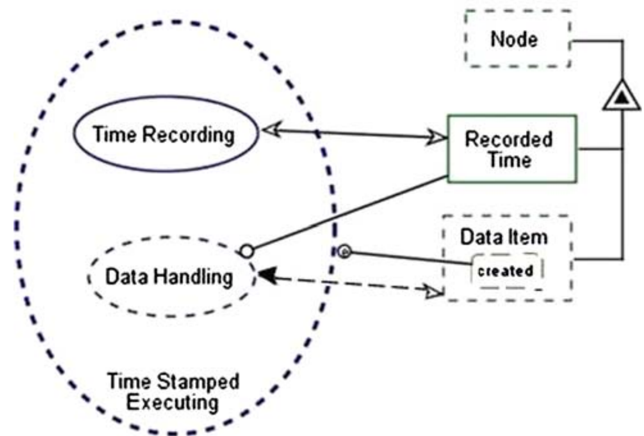

Fig. 3 A generic *Time Stamped Execution* module

target module, which is concrete and fully specified to the level required by the system designer. Generic modules are designed for reuse; hence they are the main building blocks of the OPM-based weaving process.

The weaving is done between a pair of modules, of which one is the generic module and the other is the target module. The weaving process comprises three steps: (1) designing generic and target modules, (2) integrating pairs of generic and target modules to create woven modules, and (3) enhancing the woven modules into complete systems or applications. This section explains the semantics of each of these three steps and specifies rules for checking and validating the legality of the resulting woven modules.

4.1 Designing generic OPM modules

An important attribute of any OPM element (object, process, state, or link) is its *affiliation*. The two possible values of the affiliation attribute are systemic and environmental. A *systemic element* is affiliated with (belongs to) the system. An *environmental element* is affiliated with the environment. It can be completely external to the modeled system and interact with it, or it can be partially specified and contain both environmental and systemic elements.

Figure 3 is an OPM model of a generic *Time Stamped Execution* module, which adds time recording capability to

a process execution. This module attaches to a *Data Item* a timestamp, called *Recorded Time*. *Data Item* is environmental since it needs to be concretely bound to an object of the target module in the various contexts in which the *Time Stamped Execution* module is reused. *Recorded Time*, on the other hand, is systemic since it is a local element of the generic module, and it is independent of any thing existing in the target module. Similarly, the *Time Recording* subprocess within the *Time Stamped Executing* process is systemic. *Data Handling*, which should be bound and adapted to a process in the target module, is denoted as an environmental process. The vertical layout of *Time Recording* above *Data Handling* in the context of the enclosing *Time Stamped Executing* process implies that *Time Recording* is activated first, and only after its successful termination, *Data Handling* is executed.

As noted, states and links can also be either systemic or environmental, just like objects and processes. An environmental state can be owned only by an environmental object, while an environmental (structural or procedural) link can connect only a pair of environmental entities (objects, processes, or states). An environmental state or link in a generic module requires the existence of a corresponding state or link in the target module and thus restricts the modules to which the generic module can be woven. Figure 3, for example, requires that *Data Item* objects will have at least one state, called *created*. The environmental effect link between *Data Handling* and *Data Item* in Fig. 3 means that *Data Handling* processes affect (change) *Data Item* objects.

Systemic states and links on the other hand do not impose constraints on the target modules. The systemic event link from the state *created* of *Data Item* to *Timed Stamped Executing* in Fig. 3, for example, indicates that the process is triggered each time *Data Item* enters its *created* state. In other words, whenever a new data item is created, the process that records the time of its creation is invoked.

4.2 Creating woven modules

Having obtained or created a set of generic OPM modules, the system architect should decide which ones are to be reused by weaving them into the target module, and how to weave them so that the resulting model meets the system requirements. Each (generic or target) module is enclosed in a package, which is in-zoomed to expose its inner structure and behavior. As noted, for each pair of modules to be woven, one module is defined as generic, while the other is the target. Note that a generic module in one combination can be a target in another, and vice versa. The result of weaving the generic and target modules is called a *woven module*.The woven module can be entirely concrete (i.e., all of its entities are systemic), or may still contain one or more environmental elements, implying that the weaving process has not been completed yet or that the modeled system includes external entities.

While weaving, the designer has to bind one or more environmental entities in the generic OPM module with corresponding (environmental or systemic) entities in the target module. The generalization-specialization relation is the primary means for binding an entity from the generic module to its counterpart in the target module. The generalization-specialization relation in OPM extends its object-oriented counterpart by providing not only for object inheritance, but also for process and state inheritance.

As in the object-oriented paradigm, *object inheritance* implies that the sub-object class exhibits at least the same set of features (attributes and operations) and links as the super-object class.

Process inheritance extends object inheritance to behavioral elements: the subprocess class has at least the same interface (i.e., the set of procedural links into and out of the process) and behavior (i.e., the set of subprocesses) as the super-process class. The interface and behavior of the inheriting process class may be extended. Using process inheritance, things can inherit partially-specified behaviors. This type of inheritance is difficult to achieve with an object-oriented modeling language such as UML due to the distribution of behavior specification over a number of different views, each modeling partially overlapping portions of the behavioral aspects. This situation is avoided in OPM by the recognition of Process as an important (stand-alone) concept.

In *state inheritance*, the specialized state inherits the structure (i.e., sub-states) and the interface (i.e., the set of procedural links) in which the generalized state is involved. This type of inheritance can be done also using UML Statecharts.

In Fig. 4, for example, the generic *Time Stamped Execution* module of Fig. 3 is woven into the target *product handling* module of Fig. 2, such that the combined specification, the woven module, contains two modules. Each of the three generalization-specialization relations in Fig. 4 binds an entity of the generic module to a corresponding one in the target module. These relations imply that (1) *Product Handling* inherits the systemic *Time Recording* process (which, in turn, affects *Recorded Time*), (2) *Product Updating* inherits the instrument link from *Recorded Time*, and (3) *Product Handling* inherits the event link and is thus triggered when *Data Item* is generated, i.e., when it enters its *created* state. The state generalization-specialization relation between *created Data Item* and *proper Product* implies that *proper* is a specialization of *created*, such that *Product Handling* is triggered whenever *Product* enters its *proper* state. This relation also implicitly connects *Data Item* to *Product* (implying that *Product* is a *Data Item*), as stated by the mandatory binding rule which is one of the OPM weaving rules explained next.

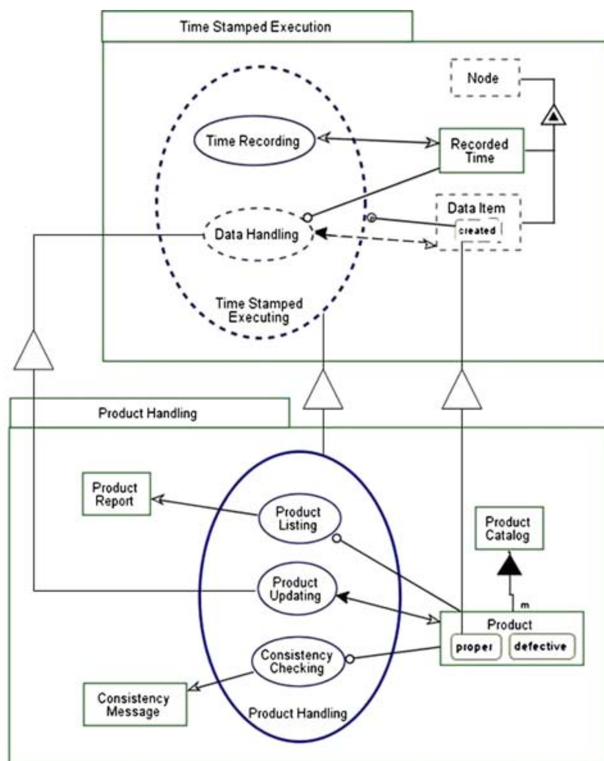

Fig. 4 An OPM woven module in which the generic *Time Stamped Execution* module (*top*) is woven into the target *Product Handling* module (*bottom*) via generalization-specialization relations

4.3 OPM weaving rules

OPM modules are required to abide by a set of weaving rules, which are divided into intra-model weaving rules and inter-model weaving rules. Intra-model weaving rules state what can and what cannot be done within a single OPM module and are elaborated in Sect. 4.3.1. Inter-model weaving rules concern the weaving of two or more modules; in particular they deal with weaving a generic module into a target module. The inter-model weaving rules are discussed in Sect. 4.3.2.

4.3.1 Intra-model weaving rules

The two OPM intra-model weaving rules are refinement and link attachment.

The refinement rule: An environmental thing in a generic module can be refined by environmental or systemic elements (objects, processes, states, and links), while a systemic thing can be refined only by other systemic elements. This is because an environmental element is more general and less restrictive than a systemic one, which is already fully specified.

An application of this rule is shown in Fig. 3: *Time Stamped Executing* must be environmental, since it zooms into (contains, or is refined to show) the environmental process *Data Handling*. Similarly, the object *Node* must be environmental, since its attribute *Data Item* is environmental. The environmental *Data Item* object owns an environmental state, called *created*.

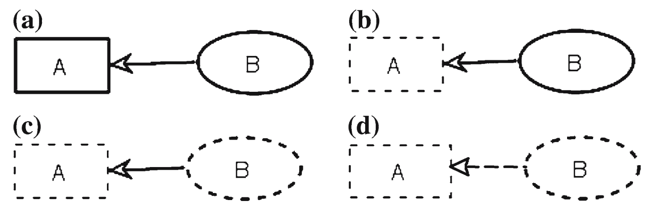

Fig. 5 The possibilities of connecting two entities in a generic OPM module: the four possible variations of linking an object and a process with a result link

The link attachment rule: In a generic module, an environmental link must connect two environmental entities, while a systemic link may connect systemic or environmental entities. This is because an environmental link imposes a requirement on the target module, so its two connected entities should appear also in the target module.

Figure 5 shows the four possible variations of linking an object and a process with a result link in a generic module. Two systemic entities or a systemic entity and an environmental one can be linked only by a systemic link (as shown in Fig. 5a, b, respectively). Two environmental entities can be linked by either a systemic or an environmental link (as Fig. 5c, d show). A systemic link between two environmental entities (as in Fig. 5c) implies that the two systemic entities in the target module, which are bound to their environmental counterparts in the generic module, must be connected after the weaving, even though this link had not been present in the original target module.

4.3.2 Inter-model weaving rules

As OPM modules, woven modules must preserve the intra-model weaving rules defined in Sect. 4.3.1. In addition, they must also follow three inter-model weaving rules, which apply to weaving a generic module with a target one: (1) mandatory binding, (2) hierarchy structure, and (3) link precedence.

The mandatory binding rule: If an environmental element (object, process, state, or link) in a generic module has a counterpart in the target module [1], then they are bound either explicitly or implicitly. Explicit binding applies a direct generalization-specialization relation. In implicit binding, the generalization-specialization relation is not visible directly. Rather, it is implied from the context, as follows:

[1] As noted, an environmental element can remain unbound in a particular binding operation, either because the binding process has not been completed yet or the envioronmental element is completely external to the modeled system.

1. Binding entities in low levels of hierarchy implicitly implies binding of entities in higher levels. For each environmental entity *A* which is refined by an environmental entity *B* in a generic module, such that *B* is explicitly bound to *Concrete B* in the corresponding target module and *A* is not bound at all, a default systemic entity, whose name is *Concrete A*, is generated in the target module. This new entity, *Concrete A*, is linked to *Concrete B* in the target module with the same type of link that *A* is connected to *B* in the generic module. In addition, *Concrete A* and *Concrete B* are linked via a generalization-specialization relation in the woven module.
2. The implicit binding of an environmental link in a generic module to a link in the target module is determined by the bindings of the entities it connects. Specifically, an environmental link in a generic module, which connects entities *A* and *B*, is implicitly bound to a link in the target module, which connects entities *Concrete A* and *Concrete B*, such that *A* is bound to *Concrete A* and *B*—to *Concrete B*.
3. Explicit binding of states induces binding of their owning objects. In other words, binding a state *concrete s* of an object *Concrete A* in the target module to an environmental state *s* of object *A* in the generic module implies that *Concrete A* is also bound to *A*.

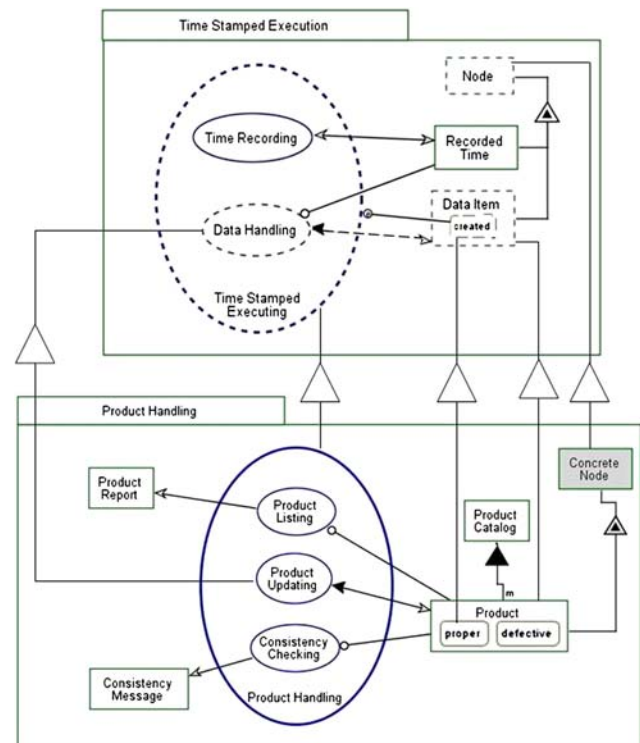

Fig. 6 The woven module of Fig. 4, in which *Concrete Node*, its binding with *Node*, and the binding of *Product* to *Data Item* explicitly appear

Examples of the above implicit binding can be found in Fig. 4. The environmental object *Node* in the generic module of this figure is not explicitly bound to an object in the target module, while its feature *Data Item* is bound to *Product*. Hence, a default systemic object, whose name is automatically set to *Concrete Node*, is generated in the target module. This *Concrete Node* object exhibits *Product* as its attribute in the target module, since *Product* is bound to *Data Item* and *Node* exhibits *Data item*. *Concrete Node* also inherits all the attributes of *Node* which are specified in the generic module.

The environmental effect link between *Data Item* and *Data Handling* in the generic module in this figure is implicitly bound to the systemic effect link between *Product* and *Product Handling* in the target module. In addition, *Product* is implicitly bound to *Data Item*, since the state *proper* of *Product* is bound to the state *created* of *Data Item*.

Figure 6 extends the woven module of Fig. 4 with the above implicit bindings. One should bear in mind that Fig. 6 is only drawn to explicitly illustrate the various bindings, but the two added bindings (one from *Time Stamped Execution* to *Product Handling* and the other from *Data Item* to *Product*) are implicitly implied from Fig. 4.

The hierarchy structure rule: The hierarchy structure of entities in a generic module must be congruent with the hierarchy structure of the corresponding entities that are bound to them in the target module.

For example, in the woven module of Fig. 6, *Node* is bound to *Concrete Node*, while *Data Item* is bound to *Product*. Hence, *Concrete Node* and *Product* of the target *Product Handling* module must preserve the exhibition-characterization relation between *Node* and *Data Item* in the generic *Time Stamped Execution* module. The same structure exists in the woven module in Fig. 4, albeit implicitly. A similar situation holds between *Product Handling*, *Time Stamped Executing*, *Product Updating*, and *Data Handling*. This time the preserved hierarchy structure is containment (in-zooming) relations.

The link precedence rule: Environmental links can be (implicitly) bound to systemic links which are at least as strong as their environmental counterparts according to the link precedence order [48]. Generally speaking, OPM's procedural link precedence order is such that the triggering links are the most powerful. These links are followed by transformation and enabling links, in this order.

As an example, the link precedence rule implies that the procedural link between *Product* and *Product Updating* in the target module in Fig. 4, which is implicitly bound to the effect link between *Data Item* and *Data Handling*, must be at least as strong as a transformation link. Hence, *Product* and *Product Updating* can be linked in the target module by a triggering or transformation link, but not by enabling links. This is due to the convention that the generic module

specifies the minimal requirements from the target modules into which it can be woven.

4.4 Generating merged modules

So far we have seen how a generic module is woven into a target module, while graphically the two modules remain separate and their bindings are maintained in the woven module. *Merging* is an alternative view of weaving. A merged module is the single module constructed by applying the weaving rules and the semantics of the woven modules defined and explained above. By definition, a woven module and its merged counterpart are semantically equivalent.

Figure 7 is the merged module derived from and equivalent to the woven module in Fig. 4. In this merged module, *Concrete Node* exhibits two attributes, *Recorded Time* and *Product*, the latter being part of *Product Catalog*. Zooming into *Product Handling* shows that it consists of four subprocesses: *Time Recording*, *Product Listing*, *Product Updating*, and *Consistency Checking*. *Product Updating*, for example, has an instrument link from *Recorded Time*, required by the generic module *Time Stamped Execution*, and an effect link with *Product*, as the target module *Product Handling* specifies. In other words, *Product Updating* uses *Recorded Time* as input and affects *Product*. The state *proper* of *Product* inherits an event link to *Product Handling* from the environmental state *created* of *Data Item* in the generic *Time Stamped Execution* module.

As noted, the time axis within each in-zoomed process is directed from the top of the diagram to its bottom. Hence, two independent or concurrent subprocesses are depicted at the same vertical level, defining a partial execution order of the subprocesses. The generalization-specialization relations between processes of different modules merge the partial orders from each module into a single combined partial order. In Fig. 4, for example, there is a total order in both the generic module *Time Stamped Execution* (first *Time Recording* and then *Data Handling*) and in the target module *Product Handling* (first *Product Listing*, then *Product Updating*, and finally *Consistency Checking*). The generalization-specialization relation between *Data Handling* and *Product Updating* in Fig. 4 defines a combined, partial order in the woven module, according to which, *Time Recording* and *Product Listing* are independently executed first, followed by *Product Updating*, and finally by *Consistency Checking*.

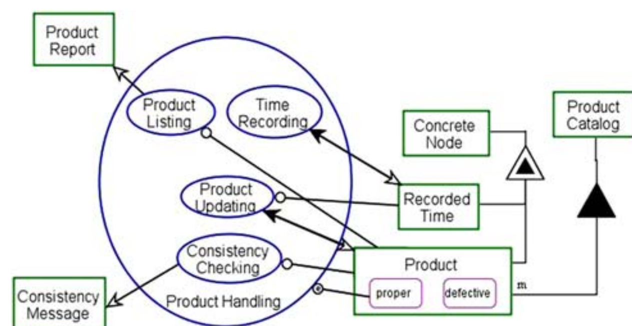

Fig. 7 The merged module which is derived from and equivalent to the woven module in Fig. 4

After merging the generic module into the target one, if more than one link exists between two entities in the merged module, then the link with the higher precedence according to the link precedence order prevails. For example, if in Fig. 4 a systemic enabling link between *Data Item* and *Data Handling* exists in the generic module, it would have been subsumed by the effect link between *Product* and *Product Updating* in the merged module, because a transformation link takes precedence over an enabling link.

4.5 Weaving versus merging

A woven module can be maintained either as is, or as a merged module. Each option has advantages and disadvantages. The woven module is more succinct and more abstract than its merged counterpart. Its main advantage is the ability to maintain and develop the generic and the target modules separately. When a generic module is improved, each target module can automatically benefit from this improvement. The main advantage of the merged module is the explicit presentation of all its elements from both the generic and the target modules in a single model. However, once merged, the module loses its linkage with the generic module, so changes made to the generic module will not be reflected in the merged module.

Modules can be maintained in libraries. Using standard facilities, such as Web services, these libraries can be searched for and distributed over many nodes (computers). Adopting a service-oriented architecture (SOA) [26,42] approach, each time an application is compiled, the most up-to-date generic modules that comprise the application are imported, thereby potentially enabling continuous improvement in various performance aspects. In another configuration, only future uses of the new version of the generic module benefit from the improvements, while existing bindings keep using the version of the module with which they were originally complied. To increase flexibility and ensure that the application enjoys potential improvements in generic library modules, one should maintain the woven module versions and generate the merged modules only to facilitate the readability of the entire application.

4.6 Enhancing woven modules

Having created a woven module from two or more modules (such that generic modules are woven one at a time),

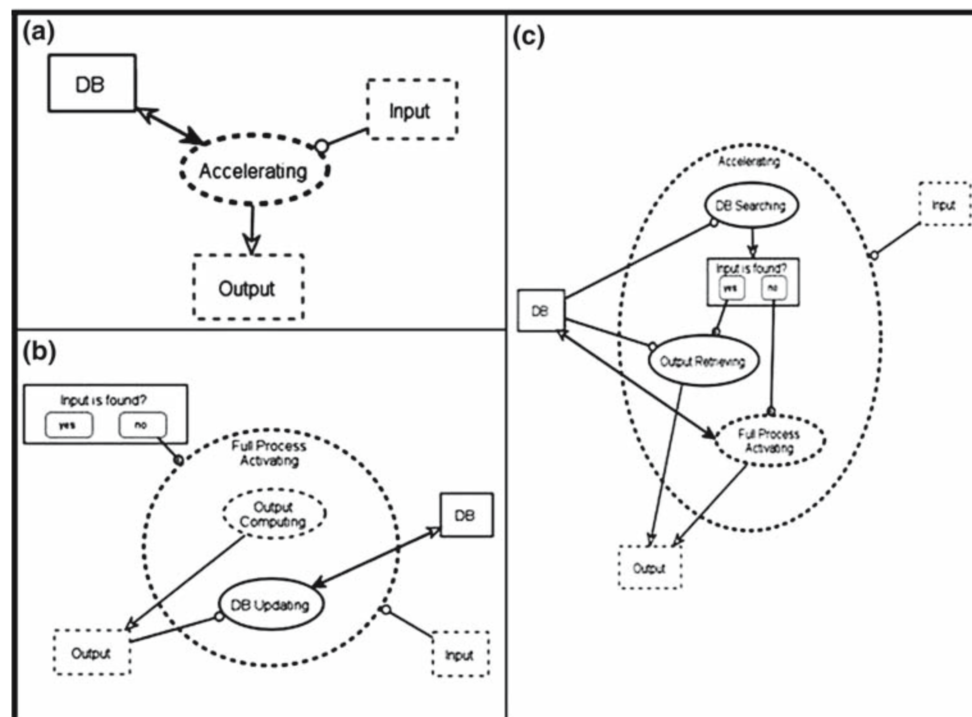

Fig. 8 The *Acceleration* module. **a** The top-level diagram. **b** *Accelerating* in-zoomed. **c** *Full Process Activating* in-zoomed

the system architect can refer to it as a single module and continue refining the system specification in a separate layer without affecting the composing generic modules. The weaving layer includes the generalization-specialization relations and possibly additional links. During the refinement stage one can enhance the combined module as part of a complete application, offering functionality that exceeds the sum of the individual module functionalities. In our example, consider a situation in which we want to use the *Recorded Time* (from the generic module) as input for *Product Listing* (from the target module), so *Product Report* will include a record of the latest time each product was updated. To achieve this, we can add an instrument (unchangeable input) link between *Recorded Time* and *Product Listing* in the woven module in Fig. 4. This instrument link, as well as the generalization-specialization relations between the two modules, is maintained in the separate layer of the woven module. The weaving process can also be successively and recursively applied to reuse additional modules for meeting new requirements or modeling various concerns or aspects.

5 Evaluation of the OPM-based weaving process

To evaluate the OPM-based weaving process presented in this paper, we used several case studies, including a Web-based accelerated search system, which we summarize in this section[2]. This system implements an algorithm for improving the performance of a Web search engine, which employs time-consuming search algorithms. The design of the Accelerated Search System included two modules — a generic *Acceleration* module and a target *Multi Search* one. The *Acceleration* module specifies a generic algorithm that reduces the execution time of an input-output part of a system by trying first to retrieve the output, which is determined by the input, from a local database. We assume that the sought Web-based items rarely change, so they are relatively static. This implies that results of subsequent activations of a query with the same input remain valid and can therefore be stored and retrieved to avoid executing the costly calculation each time a query with that input is submitted. If the entry is not already in the database, the algorithm activates a process that calculates or otherwise obtains the sought output and records it in the database to accelerate future executions of the query with the same input. Figure 8 presents the OPM model of the *Acceleration* module. Note that the in-zooming mechanism is applied in order to refine and detail processes without losing their wider context.

The *Multi Search* module implements a new search engine that benefits from existing search engines by combining their

[2] Due to space limitations, the complete specification of the Web-based accelerated search system is not given here. It can be found at http://mis.hevra.haifa.ac.il/~iris/research/WebAcceleratedSearch.pdf

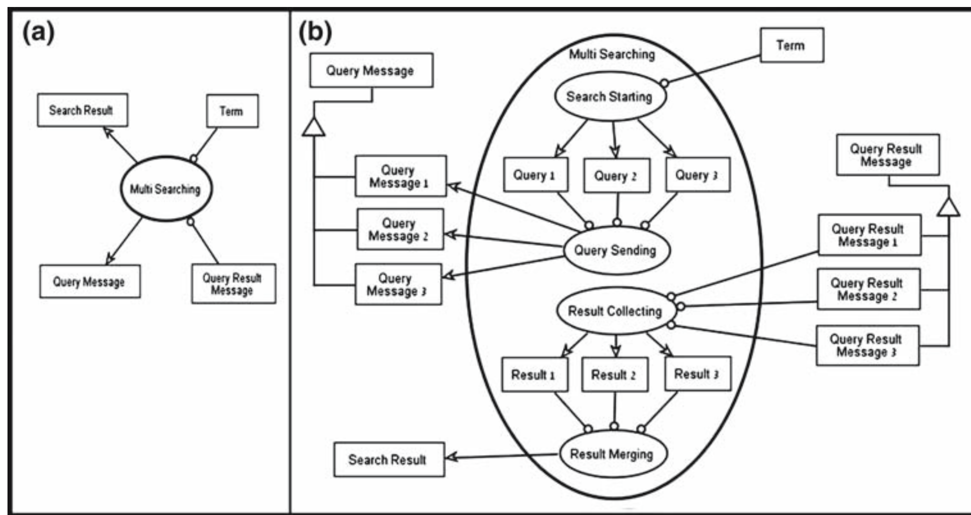

Fig. 9 The *Multi Search* module. **a** The top-level diagram. **b** *Multi Searching* in-zoomed

results and ordering them according to a weighted score. Figure 9 presents the OPM model of the *Multi Search* module.

Since the *Multi Searching* process depends on the speed of each search engine, the network response time, and the number and size of results supplied by each search engine, we weave the *Acceleration* module into the *Multi Search* module, in order for the most recently searched terms and their corresponding results to be saved in a local database. For each new query, this local database is searched before invoking the entire *Multi Searching* process, and only if the result is not found there, the system will execute the *Multi Searching* process. Figure 10 shows the generic *Acceleration* module, from Fig. 8c, woven into the *Multi Search* module, from Fig. 9a. We selected specifically these two diagrams as they best serve our weaving purposes. The resultant woven module, called *Accelerated Multi Search*, includes three generalization-specialization relations that connect the things in the generic module to the corresponding things in the target module. Two of these relations are between object classes, specifying that *Term* is an *Input* and that *Search Result* is an *Output*. The third generalization-specialization relation is between two process classes, specifying that the systemic *Multi Searching* process specializes the environmental *Output Computing* process.

We found that, in comparison to the object-oriented approach in general and UML in particular, our approach expresses more naturally and comprehensively the functionality of the generic, target, and woven modules. This, in turn, stems from OPM's recognition of process classes as first-class citizens beside object classes rather than as just methods encapsulated within object classes. Although some form of process inheritance is available in UML, since behaviors can be regarded as classes, the exact functionality of the processes cannot be described this way, and partially specified behaviors, which are naturally expressed in OPM, can hardly be supported in UML. Furthermore, in order to express the models in Figs. 8–10 in UML, one needs to employ UML use case, class, sequence, and Statechart diagrams, requiring application of consistency rules among the various views.

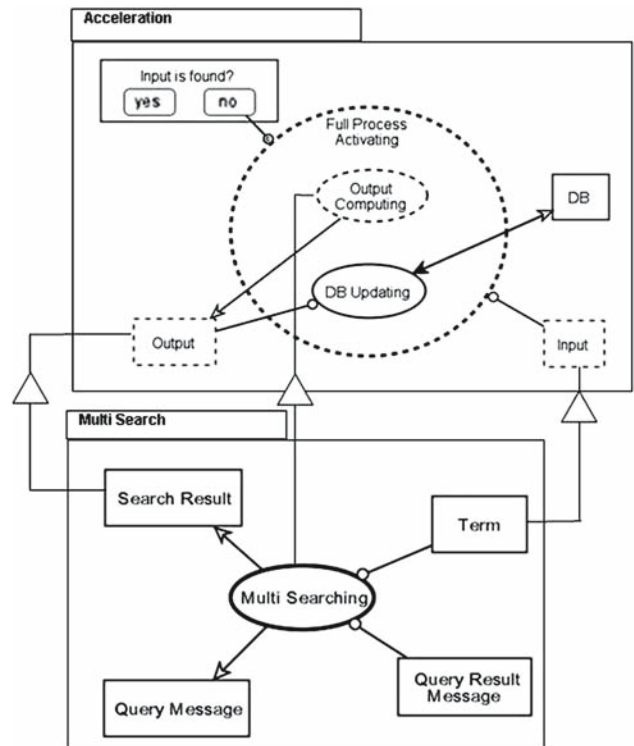

Fig. 10 The *Accelerated Multi Search* module

Furthermore, the refinement-abstraction mechanisms within OPM in general and the in-zooming mechanism in particular enable designers to specify generic modules that are more detailed than commonly used design patterns. While adding details to generic modules might reduce the number of contexts in which the module can be reused, it also provides a solid template to work with in detailed design phases.

The equivalent semantics of the woven and the merged modules enables treating woven modules as either generic or target OPM modules in other weaving operations. The system architect can continue specifying the system into which a module has been woven as a complete application. For example, in the woven *Accelerated Multi Search* module, the *Result Merging* algorithm within *Multi Searching* can be improved by treating *DB* as an additional input and/or an entire generic *Log Recording* module can be introduced[3]. Any combination of these two refinements can be part of the system design.

6 Discussion and future work

We have presented a reuse approach that applies weaving of generic and target modules expressed in a single modeling paradigm that combines structure and behavior by presenting and linking objects and processes in the same single diagram type. We distinguish between systemic elements that are fully specified and environmental elements which need more specification in the target context. Our approach enables the construction of a system model from modules that can be either woven or merged. We preferred OPM over UML as the basis of our work due to its underlying ontology that allows expressing intricate models that combine structure and behavior and provide for expressing both generic and target modules. By representing structure and behavior in a single diagram type, OPM enables reusing behavioral modules and organizing their dynamic aspects, such as ordering, synchronization, etc., into complete applications. Augmentations that otherwise cut across class boundaries and diagram types are described naturally in a single, generic OPM module. This is done with a relatively small vocabulary of elements, greatly reducing the number and severity of consistency and integration problems, which are known to be a notorious hindrance in multi-view modeling languages [45,51,52]. However, some of the ideas in this paper can be incorporated into the UML framework. For example, state inheritance can be introduced to UML statechart diagrams, while environmental vs. systemic elements can be specified by introducing UML stereotypes.

The approach to reuse presented and demonstrated in this paper advocates maintaining the reusable modules at a high level of abstraction and refining them in specific contexts. The OPM combination of object- and process-oriented paradigms enables modeling generic behaviors that cut across system structures and adapting them to specific target modules in a clean and clear way, making them ideal candidates for enhancing aspect-oriented modeling [2]. A set of inter- and intra-weaving rules determines how to define and how to combine reusable generic modules. The process starts with developing generic and target modules. It continues with successively weaving generic modules into the current target module by binding elements in the target module to corresponding elements in the generic module via generalization-specialization links. The resulting combined model can be further refined into a complete, fully functional application.

Such reuse is transparent, or white-box, rather than black-box: the generic modules can be modified internally to improve various performance criteria while maintaining their interface and intended functionality. This way one can enjoy the best of both worlds: adapt the model to meet specific requirements of the system under development while taking advantage of continuously improving generic modules that reflect current, constantly updated best practices, which can be dynamically woven into the applied system.

As a graphical notation, OPM does not presently support logical (textual) constraints. In order to achieve abstraction, which is an important key for reuse, the modeling language should support such constraints and not just visually expressed functional and structural descriptions. Constraints can be used at the process, object, or module level to specify functionality requirements from an entity. Although not presently supported in OPM, the Object Constraint Language (OCL) [40] can be applied as a complementary tool for this purpose (similarly to the usage of OCL within UML).

Work is under way to support the weaving process described in this work into Object-Process CASE Tool (OPCAT) [16], an integrated software engineering environment, which enables specifying systems in OPM and simulating them at the design level. In particular, OPCAT includes an OPM-GCG component, which is a generic code generator that translates OPM design models to various target programming languages, including object-oriented ones, such as Java. The translation rules are defined offline through a user-friendly tool and can be changed by the designer. These rules are used by the implementation generator to create code files from the OPM specification of the merged modules. Our experience, which is elaborately reported in [46], shows that the Java code generated by OPM-GCG includes system behavior (processes, control flows, event triggers, etc.), as well as structure. Furthermore, comparing this code to the code generated from a leading UML CASE tool for analogous UML models, the OPM-GCG code appears to be

[3] These examples can be found at http://mis.hevra.haifa.ac.il/~iris/research/WebAcceleratedSearch.pdf

simpler, more intuitive, easier to maintain and update, and more complete, while being almost three times as short as the code generated by the chosen UML CASE tool. We plan to introduce into the OPM-GCG rules for translating woven modules into aspect-oriented programming languages, such as AspectJ [3], Timor [30], Contexts in .NET [34], or the container/server model of EJB [35].

References

1. Aldawud, O., Elrad, T., Bader, A.: A UML Profile for Aspect Oriented Modeling. Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, (2001). Available at http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/submissions/26-aldawud.pdf
2. The Aspect-Oriented Software Development Web site. http://www.aosd.net/
3. AspectJ Web Site. http://www.eclipse.org/aspectj/
4. Baniassad, E., Clarke, S.: Theme: an approach for aspect-oriented analysis and design. 26th International Conference on Software Engineering (ICSE 2004), IEEE Computer Society, 2004, pp. 158–167 (2004)
5. Bouge, L., Francez, N.: A Compositional Approach to Superimposition. Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, 1998, pp. 240–249 (1998)
6. Barber, K.S., Graser, T.J., Jernigan, S.R.: Increasing Opportunities for Reuse through Tool and Methodology Support for Enterprise-wide Requirements Reuse and Evolution. Proc. of the 1st International Conference on Enterprise Information Systems, 1999, pp. 383–390 (1999)
7. Back, R.J.R., Kurki-Suonio, R.: Decentralization of process nets with centralized control. Distrib. Comput. **3**, 73–87 (1989)
8. Booch, G.: Object-oriented analysis and design with application. Benjamin/Cummings Publishing Company, Inc. (1994)
9. Bosch, J.: Superimposition: a component adaptation technique. Inf. Softw. Tech. **41**(5), 257–273 (1999)
10. Constantinides, C.A., Bader, A., Elrad, T.: An Aspect-Oriented Design Framework for Concurrent Systems. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1999, pp. 340–352
11. Clarke, S.: Extending standard UML with model composition semantics. Sci. Comput. Program. **44** (1), 71–100 (2002). http://www.cs.tcd.ie/people/Siobhan.Clarke/papers/SoCP2001.pdf
12. Clarke, S., Baniassad, E.: Aspect-oriented analysis and design: the theme approach. The Addison-Wesley Object Technology Series (2005)
13. Clarke, S., Walker, R.J.: Composition patterns: an approach to designing reusable aspects. Proceedings of the International Conference on Software Engineering, pp. 5–14 (2001)
14. Dori, D.: Object-process methodology—a holistic systems paradigm. Springer, Heidelberg (2002)
15. Dori, D.: Why significant UML change is unlikely. Commun. ACM **45**(11), 82–85 (2002)
16. Dori, D., Reinhartz-Berger, I., Sturm A.: OPCAT — A Bimodal Case Tool for Object-Process Based System Development. 5th International Conference on Enterprise Information Systems (ICE-IS 2003), 2003. Software download site: http://www.opcat.com/
17. D'Souza, D., Wills, A.C.: Objects, frameworks and components with UML — the catalysis approach. Addison-Wesley, Reading (1998)
18. Eckstein, S., Ahlbrecht, P., Neumann, K.: Increasing Reusability in Information Systems Development by Applying Generic Methods. Proceedings of the 13th International Conference CAiSE'01, LNCS 2068, 2001, pp. 251–266 (2001)
19. Early Aspect website: Aspect-Oriented Requirements Engineering and Architecture Design. http://www.early-aspects.net/
20. Frakes, W., Terry, C.: Software reuse: metrics and models. ACM Comput. Surv. **28**(2), 415–435 (1996)
21. Gomma, H.: Designing software product lines with UML: from use cases to pattern-based software Architectures, Addison Wesley, Reading (2004)
22. Gamma, E., Helm, R., Johnson, R. Vlissides, J.O.: Design patterns. Addison-Wesley, Reading (1995)
23. Griss M., Favaro J., d'Alessandro M. Integrating Feature Modeling with the RSEB, Proceedings of the Fifth International Conference on Software Reuse, pp. 76–85 (1998). http://www.favaro.net/john/home/publications/rseb.pdf
24. Grundy, J.: Multi-perspective specification, design and implementation of software components using aspects. Int. J. Softw. Eng. Knowl. Eng. **10**(6), 713–734 (2000)
25. Gurp, J.V., Bosch, J., Svahnberg, M.: On the Notion of Variability in Software Product Lines, Working IEEE/IFIP Conference on Software Architecture (WISCA'01), 2001, pp. 45–54 (2001)
26. IBM Corp. SOA and Web Services. http://www-106.ibm.com/developerworks/webservices/newto/
27. Kande, M.M.: A concern-oriented approach to software architecture. Computer Science, vol. PhD. Lausanne, Switzerland: Swiss Federal Institute of Technology (EPFL) (2003). Available at http://biblion.epfl.ch/EPFL/theses/2003/2796/EPFL_TH2796.pdf
28. Katara, M., Katz, S.: A concern architecture view for aspect-oriented software design. Software and system modeling, Springer, (2006). doi:10.1007/s10270-006-0032-x
29. Katz, S.: A superimposition control construct for distributed systems. ACM Trans. Program. Lang. Syst. **15**(2), 337–356 (1993)
30. Keedy, L., Heinlein, C., Menger, G.: The Timor Programming Language. http://www.jlkeedy.net/timor-programming_language.html
31. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., M., Irwin, J. Aspect-Oriented Programming. European Conference on Object-Oriented Programming (ECOOP'97). LNCS 1241, pp. 220–242 (1997)
32. Kulesza, U., Garcia, A., Lucena, C. Towards a method for the development of aspect-oriented generative approaches, Early Aspects workshop, OOPSLA'2004 (2004). Available at http://trese.cs.utwente.nl/workshops/oopsla-early-aspects-2004/Papers/KuleszaEtAl.pdf
33. Lester, N.G., Wilkie, F.G., Bustard, D.W.: Applying UML Extensions to Facilitate Software Reuse. The Unified Modeling Language (UML'98) - Beyond the Notation. LNCS 1618, pp. 393–405 (1998)
34. Lowy, J.: Contexts in .NET: Decouple Components by Injecting Custom Services into Your Object's Interception Chain, MSDN Magazine—The Microsoft Journal for Developers (2003). Available at http://msdn.microsoft.com/msdnmag/issues/03/03/ContextsinNET/#S1
35. Mahapatra, S.: Programming restrictions on EJB, Java World (2000). Available at http://www.javaworld.com/javaworld/jw-08-2000/jw-0825-ejbrestrict.html
36. Mapelsden, D., Hosking, J., Grundy, J.: Design Patterns Modelling and Instantiation using DPML. 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS), (2002). http://www.jrpit.flinders.edu.au/confpapers/CRPITV10Mapelsden.pdf
37. Mens, T., Lucas, C., Steyaert, P.: Giving Precise Semantics to Reuse and Evolution in UML. Proc. PSMT'98 Workshop on Precise Semantics for Modeling Techniques (1998)

38. Mezini, M., Lieberherr, K.: Adaptive Plug-and-Play Components for Evolutionary Software Development. Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 97–116 (1998)
39. Mili, H., Mili, F., Mili, A.: Reusing software: issues and research directions. IEEE. Trans. Softw. Eng. **21**(5), 528–562 (1995)
40. Object Management Group: Object Constraint Language, Version 2.0 (2005)
41. Object Management Group: The Unified Modeling Language (UML™), version 2.0 (2005)
42. Papazoglou, M.P.: Service-oriented computing: concepts, characteristics and directions, Proceedings of the 4th IEEE International Conference on Web Information Systems Engineering (WISE'2003), pp. 3–12 (2003)
43. Peleg, M., Dori, D.: Extending the Object-Process Methodology to handle real-time systems. J. Object. Oriented. Program. **11**(8), 53–58 (1999)
44. Peleg, M., Dori, D.: The model multiplicity problem: experimenting with real-time specification methods. IEEE Trans. Softw. Eng. **26**(8), pp. 742–759 (2000). http://iew3.technion.ac.il:8080/Home/Users/dori/Model_Multiplicity_Paper.pdf
45. Reinhartz-Berger, I.: Conceptual Modeling of Structure and Behavior with UML—The Top Level Object-Oriented Framework (TLOOF) Approach, the 24th International Conference on Conceptual Modeling (ER'2005), Lecture Notes in Computer Science 3716, pp. 1–15 (2005)
46. Reinhartz-Berger, I., Dori, D.: Object-Process Methodology (OPM) vs. UML: A Code Generation Perspective, Evaluating Modeling Methods for System Analysis and Design (EMMSAD'04), Proceeding of CAiSE'04 Workshops, vol. 1, 1004, pp. 275–286
47. Reinhartz-Berger, I., Dori, D.: OPM vs. UML—Experimenting with Comprehension and Construction of Web Application Models. Empir. Softw. Eng. **10**(1), 57–80 (2005)
48. Reinhartz-Berger, I., Dori, D.: A reflective metamodel of object-process methodology. In: Green, P., Rosemann, M., (eds.) The System Modeling Building Blocks, chapter 6 in Business Systems Analysis with Ontologies, Idea Group Inc., (2005)
49. Reinhartz-Berger, I., Dori, D., Katz, S.: OPM/Web – Object-Process Methodology for Developing Web Applications. Annals on Software Engineering – Special Issue on Object-Oriented Web-based Software Engineering, pp. 141–161 (2002)
50. Schauerhuber, A., Schwinger, W., Kapsammer, E., Retschitzegger, W., Wimmer M., Kappel, G. A Survey on Aspect-Oriented Modeling Approaches (2006). Available at http://www.bioinf.jku.at/publications/2006/1506.pdf
51. Siau, K., Cau, C.: Unified modeling language: a complexity analysis. J. Database Manag. **12**(1), 26–34 (2001)
52. Skipper, J.F.: Assessing the Suitability of UML for Capturing and Communicating System Engineering Design Models. Vitech Corporation, 2002. http://www.vitechcorp.com/infocenter/SuitabilityOfUMLForSE_2002.pdf
53. Soffer, P., Golany, B., Dori, D., Wand, Y.: Modelling Off-the-shelf information systems requirements: an ontological approach. Requir. Eng. **6**(3), 183–199 (2001)
54. Special Issue on Aspect-oriented Programming, Communication of the ACM, 44 (10), 2001
55. Stein, D., Hanenberg, S., Unland, R.: A UML-based Aspect-Oriented Design Notation for AspectJ. Proceedings of the 1st International Conference on Aspect-Oriented Software Development, ACM, pp. 106–112 (2002)
56. Sunye, G., Le Guennec, A. Jezequel, J.M.: Design Patterns Application in UML. Proceedings of the 14th European Conference on Object-Oriented Programming, LNCS 1850, pp. 44–62 (1850)

Author's Biography

Iris Reinhartz-Berger is a faculty member at the Department of Management Information Systems, Haifa University, Israel. She received her BSc degree in applied mathematics and computer science from the Technion, Israel Institute of Technology in 1994. She obtained a MSc degree in 1999 and a PhD in 2003 in information management engineering from the Technion, Israel Institute of Technology. Her MSc and PhD dissertations dealt with improving various development phases, mainly design and implementation, in Object-Process Methodology (OPM). Her research interests include conceptual modeling, modeling languages and techniques for analysis and design, domain analysis, development processes, and methodologies. Her work has been published in international journals and conferences.

Dov Dori is Head of the Information Systems Engineering Area at the Faculty of Industrial Engineering and Management, Technion, Israel Institute of Technology, and Research Affiliate at Massachusetts Institute of Technology. His research interests include Complex Systems Modeling, Systems Engineering and Architecture, Software Engineering, and Information Systems Engineering. Prof. Dori has developed Object-Process Methodology (OPM), a holistic systems paradigm for conceptual modeling, presented in his 2002 book (by Springer). Prof. Dori has won the Technion Klein Award for OPM and the Hershel Rich Technion Innovation Award for OPCAT, the OPM supporting software. Prof. Dori authored six books and over 100 journal publications and book chapters. He is Fellow of the International Association for Pattern Recognition and Senior Member of IEEE and ACM.

Shmuel Katz received his Ph.D. from the Weizmann Institute of Science. He heads the Software Engineering Track of the Computer Science Department of the Technion–Israel Institute of Technology. He has written over 70 journal and conference papers on program verification, specification, and methodology. His research interests include aspect-oriented programming and software development, program verification, partial order reductions in verification, and translations among verification and modeling tools. He is the head of the Formal Methods Lab of the AOSD-Europe Network of Excellence, coordinating work on formal methods and semantics for aspects.