Interner Bericht

MVP-L Language Report Version 2

Alfred Bröckers
Christopher M. Lott
H. Dieter Rombach
Martin Verlage

265 / 95

Fachbereich Informatik

Universität Kaiserslautern • Postfach 3049 • D-67653 Kaiserslautern

MVP-L Language Report Version 2

Alfred Bröckers
Christopher M. Lott
H. Dieter Rombach
Martin Verlage

265 / 95

Herausgeber: AG Software Engineering
Leiter: Prof. Dr. H. Dieter Rombach

Kaiserslautern, February 1995

Abstract

Intellectual control over software development projects requires the existence of an integrated set of explicit models of the products to be developed, the processes used to develop them, the resources needed, and the productivity and quality aspects involved. In recent years the development of languages, methods and tools for modeling software processes, analyzing and enacting them has become a major emphasis of software engineering research. The majority of current process research concentrates on prescriptive modeling of small, completely formalizable processes and their execution entirely on computers. This research direction has produced process modeling languages suitable for machine rather than human consumption. The MVP project, launched at the University of Maryland and continued at Universität Kaiserslautern, emphasizes building descriptive models of large, real-world processes and their use by humans and computers for the purpose of understanding, analyzing, guiding and improving software development projects. The language MVP-L has been developed with these purposes in mind. In this paper, we motivate the need for MVP-L, introduce the prototype language, and demonstrate its uses. We assume that further improvements to our language will be triggered by lessons learned from applications and experiments.

This report describes the version of MVP-L as of December 1994. Examples published before December 1994 refer to earlier versions of the language and are not entirely consistent with this language report. Future publications will be based on this version of MVP-L.

Contents

List of Figures

List of Tables

Preface

This document describes the software process modeling language MVP-L. This language is intended to allow project members to describe their particular views of software development projects and processes. Descriptive modeling is motivated by the need to improve both the product and the process.

MVP-L has evolved over time. Feedback from real-world applications of the language resulted in several improvements. Many people contributed to the current version of the language, as follows:

> H. Dieter Rombach and John Marsh developed version 0.8 of the syntax at the University of Maryland in 1988.
>
> H. Dieter Rombach and Christopher M. Lott formulated version 0.9 of MVP-L in 1990 [26].
>
> Alfred Bröckers, Christopher M. Lott, H. Dieter Rombach, and Martin Verlage developed the version 1.0 of MVP-L at Universiät Kaiserslautern in 1992 [8].
>
> This report is the immediate successor of [8] and describes version 2.0 of MVP-L.

This report is intended to serve as a reference to MVP-L. It is neither a textbook on an introductory level nor a cookbook to provide solutions for a software process modeling problem. Moreover, the reader should not expect to understand MVP-L completely simply by reading this report. Experience is required to understand the *why* of particular language constructs.

The current version of the language language reflects our current experience with process modeling. All suggestions for improvement are welcome.

Alfred Bröckers, Christopher M. Lott, H. Dieter Rombach, and Martin Verlage,

February 1995

1 Introduction

Many processes by which software is developed and maintained are informal. The processes are based on implicit models which are difficult to exchange between people, to tailor to changing project goals and project environment characteristics, to analyze in advance of execution, to transfer into new projects, to control during execution, and to improve. Explicit process models are needed - in addition to explicit product and quality models - to gain intellectual control over software projects. It should be noted that many of the problems associated with implicit process models can be hidden in stable development organizations for a long time. However, they typically surface as soon as organizations experience significant degrees of personnel fluctuation (i.e., carriers of implicit process knowledge are lost) or dramatic changes in their application domains (i.e., implicit assumptions become invalidated).

Many isolated project aspects such as overall life-cycles, schedules, resource allocation schemes, milestones, or qualities of the resulting products are routinely modeled today. Problems associated with most state-of-the-practice languages designed or used for software process modeling include the inability to describe the relationships between multiple project aspects, the lack of formal execution models, and the lack of human comprehensibility. It is difficult to perform cause-effect analyses or to predict the impact of changing one aspect without understanding the relationships between different project aspects. For example, one needs to understand the relationship between the life-cycle, product models, and quality models used in a project to identify a) the life-cycle phases responsible for inadequate product quality or b) to predict the impact of changing the life-cycle model on the resulting product quality. Without a formal execution model it is difficult to perform analyses during project planning. For example, one needs to have a formal notion of execution in order to analyze given models of life-cycle, schedule, and resource allocation for consistency. Without comprehensible representations, it is difficult for humans to validate descriptive models. The formality and comprehensibility requirements may seem contradictory at first. However, we strongly believe that formal execution semantics and natural representation can coexist. In summary, we need notations capable of addressing project aspects in-the-large in a human comprehensible form yet based on a formal execution model.

Only recently has attention focused on the development of formal process models [25]. Formal process models are expected to make it easier to understand processes, to analyze and package them, to transfer them across projects, to guide the execution of software processes, and to improve the models. Much process research in the late 1980's concentrated on building prescriptive models of small processes which are entirely executable on computers. Examples include the creation of environments based on rule-based process models describing the interaction between tools [14]. Another example is the completely machine executable representation of process models in an Ada-style notation [22]. At the University of Maryland, empirical software process, product and quality models have been built from a variety of perspectives for the purposes of understanding and improvement [12]. In the TAME project, the quality improvement paradigm has been devised to create empirically validated models, package and store them, and use the models to improve future projects [1], [4], [5]. The Multi-View Process modeling (MVP) project focuses on process models, their representation, their modularization according to views, and their use in the context of the quality improvement paradigm. The process modeling language MVP-L was developed to help build descriptive process models, package them for reuse, integrate them into prescriptive project plans, analyze project plans, and use these project plans to

guide future projects. In this report, we introduce the goals of the MVP project in Section 2, and derive requirements for process modeling languages that support these goals in Section 3. The presentation of the process modeling language MVP-L at a glance, and the evaluation whether requirements have been met is done in Section 4. A detailed introduction of our prototype process modeling language is performed in the sections 5 and 6. Section 7 concludes this report and makes suggestions about future research efforts. Section 8 describes the changes that have been made with respect to earlier versions of MVP-L. Finally, Appendix A demonstrates MVP-L's usefulness by way of an example.

2 MVP Project Goals

The goals of the MVP project can be best described in the context of the improvement-oriented TAME environment model shown in Figure 1.

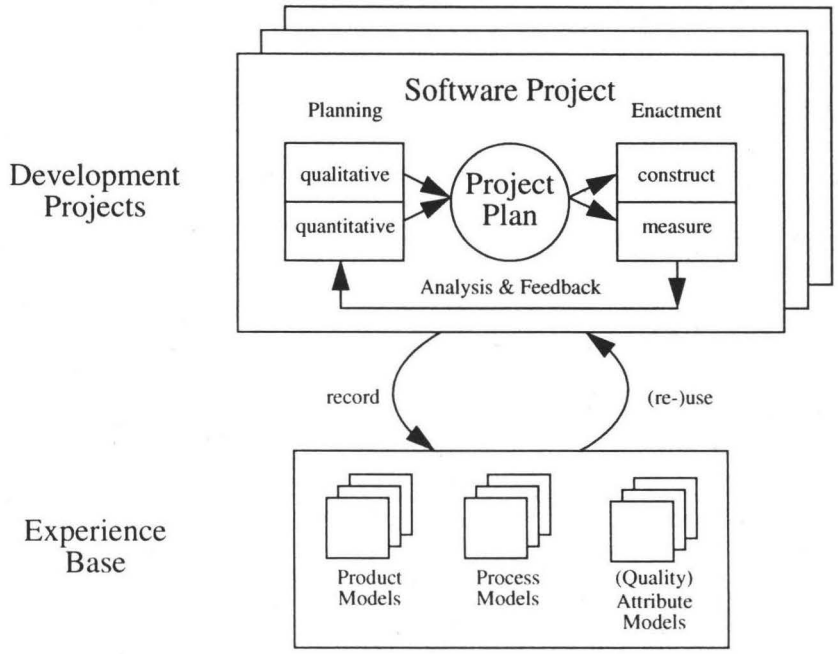

Figure 1: Improvement-Oriented TAME Environment Model

This model was developed as part of the TAME project at the University of Maryland [5]. It identifies three major project activities: planning, enactment, and analysis and feedback. Planning involves the selection and/or creation of explicit product, process, and quality models[1] and their instantiation into a project plan best suited to achieve the project goals in a given project environment. Enaction[2] involves the performance of the activities described in the project plan. Analysis and feedback involve the analysis of the collected measurement data and allow using the data to guide projects, control adherence to the project plan, trigger and guide necessary re-planning activities, and ultimately improve the project plan for future projects.

There is a growing consensus that more formal process models are needed in order to support all the activities in the TAME model [25]. There is also agreement that only the integration of construction and measurement will provide the desired engineering control for project-specific process execution and organization-specific improvement. There is, however, no consensus as to how such process models should be represented. This immaturity of the field of process model representation requires an experimental approach for the purpose of learning. The MVP project at the

1. The term 'model' is used to refer to type descriptions. A process model describes properties common to a class of processes.

2. Because processes are also performed by humans, the term *execution* is seldom used in the process community. Instead the term *enactment* is used for process performance by human agents. Only when it is clear that a machine is performing the process, the term execution is used.

University of Maryland addresses process modeling within the improvement-oriented TAME environment model. The main MVP project goal is to provide support for:

G_1. **Building descriptive process models**: Process research should start by gaining an understanding of existing processes. Descriptive modeling requires the ability to represent knowledge from the application domain 'software process' in a form which can be reviewed and validated by the application experts.

G_2. **Instantiating process models into project plans**: A project plan prescribes the products to be produced, the processes to be performed, and the resources assigned to perform processes. Objects representing products, processes, or resources are to be created and set up with initial values. Project plans need to support this instantiation of processes for execution by humans or machines.

G_3. **Packaging project plans for reuse**: Real-world projects are complex and unique. Nevertheless, they share many basic processes. Project plans (or parts of them) need to be re-packaged in a way which enables their easy modification and reuse in future projects.

G_4. **Analyzing project plans**: Analysis of a software project plan prior to enactment is as important as the analysis of a product before its operational use. Static and dynamic analyses are required. Dynamic analyses require a formal enactment model and the ability to instrument processes for data collection.

G_5. **Guiding the enactment of project plans**: Large projects are performed by groups of people performing activities in the context of different roles. Support is required for guiding and coordinating the work of people within the context of a project plan based on a formal project plan enactment model.

G_6. **Documenting project plan enactment histories**: Typical project plans allow a variety of actual project enactments. Support for documenting project enactments is required in order to perform cause-effect analyses in the case of failure, to learn how the existing model could be improved, and to simulate through past projects for the purpose of training.

The first four goals are oriented towards planning, and the latter two are oriented towards execution.

3 Process Modeling Language Requirements

Process modeling languages supporting the six goals of section 2 must allow the representation of different kinds of models, the representation of process models, their integration and instantiation into project plans, and the construction of complex process models from elementary ones. The following seven language requirements reflect these needs.

Requirements regarding the general language issues:

R_1. **natural representation**: Language features are needed to describe all relevant aspects of software projects in a notation easily understood ("natural") by humans.

R_2. **representation of different kinds of elementary models**: Language features are needed to represent different kinds of elementary models. Elementary process models are not sufficient. We cannot describe processes without referring to the products they consume and produce, or the resources (e.g., humans or machines) which perform them.

Requirements regarding the representation of elementary process models:

R_3. **use of formal process model interface parameters**: The entire interface of processes needs to be described in terms of formal parameters. This enables the instantiation of processes in changing project plan environments.

R_4. **instrumentation of processes for data collection**: Quality models, which define attributes of processes, products, and resources, are used to describe and control the behavior of processes. The data needed to derive an attribute's value is collected during the enactment of the processes. Language features are needed to describe data collection in the processes.

Requirements regarding the use of elementary process models in project plans:

R_5. **instantiation of processes**: Process objects are instantiated from process models in the context of project plans. Language features are required to support the instantiation of processes and how to relate the process instances.

R_6. **formal process execution model**: Processes are intended for enactment. Language features must be formulated in a form understandable by a machine.

Requirements regarding the construction of complex process models:

R_7. **modularization of process models**: Complex processes are composed from elementary ones. Language features are needed to describe the decomposition of complex process models into elementary ones and vice versa.

Table 1 summarizes the most obvious relationships between these seven language requirements (R_i) and the six goals (G_i) from Section 2.

LANGUAGE REQUIREMENTS	MVP PROJECT GOALS					
	G_1	G_2	G_3	G_4	G_5	G_6
R_1 (natural representation)	X	X	X	X	X	X
R_2 (different kinds of elementary models)	X	X	X	X	X	X
R_3 (formal parameters)	X		X			
R_4 (instrumentation for data collection)	X			X		
R_5 (instantiation of processes)		X				
R_6 (formal execution model)		X		X	X	X
R_7 (modularization)			X			

Table 1: Requirements-Goals Matrix

For example, if a process modeling language fulfils R_1 this supports all the goals we identified for software process modeling. A natural representation eases the mapping between real-world phenomena and modeled objects (G_1 <-> R_1) and allows a re-mapping of the objects of a project plan onto real-world objects (G2 <-> R_1). R_2 is important as R_1, because it also contributes to the entire set of goals. This is not surprising, because both requirements state that the set of basic language features should be complete, reflect real-world concepts, and rich.

The row belonging to R_6 states that a formal definition of a process modeling language supports the instantiation of process models (R_6 <-> G_2) for example by defining exactly process data, supports analyzing project data (R_6 <-> G_4) for example by providing information about process enactment for deadlock detection algorithms, guiding project people in the project (R_6 <-> G_5) for example by automatically updating the project state, and documents project history (R_6 <-> G_6) for example by interpreting sequences of events.

The assessment of eighteen process representation languages during the 6th International Software Process Workshop revealed that only two other languages - MERLIN [24] and Statemate [15] - can be considered viable candidates for descriptive modeling according to our requirements [16]. Both are explicitly aimed at human understanding. MERLIN is a rule-based language based on PROLOG. Statemate is a modeling technique based on Statecharts and was originally designed to model real-time system requirements [13]. Processes are modeled in Statemate from three perspectives: functional, behavioral, and organizational. We felt that neither of these two languages completely satisfies our seven requirements. MERLIN does provide a formal execution model (R_6). However, we are not satisfied with MERLIN's approaches, or lack thereof, to natural modeling (R_1), to modularization (R_7), to formal interface representation (R_3), and to process instrumentation (R_4). Statemate does provide a view-based approach to modularization (R_7). However, we are not satisfied with the fact that perspectives cannot be defined by the modeler, and with Statemate's approaches, or lack thereof, towards formal interface

representation (R_3), and process instrumentation (R_4). Both MERLIN and Statemate contain many useful features for process modeling. Overall, however, we felt we needed a new language with more emphasis on our requirements regarding natural modeling (R_1), explicit formal interface representation (R_3), process instrumentation (R_4), and flexible modularization (R_7).

A more detailed discussion of several approaches for process modeling can be found in [30]. A schema for evaluating a language from a personal point of view is presented in [29].

4 Introduction to the Process Modeling Language MVP-L

We introduce the process modeling language MVP-L in terms of its underlying philosophy, language definition, and language implementation. This chapter should give an overview about MVP-L. Detailed information is presented in the subsequent chapter.

4.1 Philosophy

The main focus of MVP-L is on modeling in-the-large. We assume that the payoff from being able to understand, guide, and support the interaction between individual processes will be bigger than the payoff from completely automating low-level activities routinely performed by individuals.

Process modeling in-the-large and process modeling in-the-small are analogous to programming in-the-large and in-the-small. There should be evolution of process representations on several levels of abstraction in order to become explicit process models that are complete, detailed for enactment, maintainable, reusable, aso. Neither fine-grain process programs nor coarse-grain process models are suitable for achieving all goals. This again directly corresponds to the product world where representations of the software system exist on various levels of abstraction, too. We are aware that other languages for process definition should complete MVP-L.

4.2 Overview

The MVP-L language is designed to model processes, products, resources, and quality attributes, and support their instantiation in project plans. A process model is actually a type description which captures the properties common to a class of processes. In order to enable the easy adaption of processes to changing project contexts, each process model is separated into a description of its interface and its body. The description of the interface is based on a generalization of the formal parameter concept to all kinds of models. For example, the process model 'Design' in Figure 2 describes a class of processes which require as input a product of type 'Requirements_document', must produce as output a product of type 'Design_document', and must be executed by a resource of type 'Design_group'. MVP-L supports strong typing enabling interface checking at compile time.

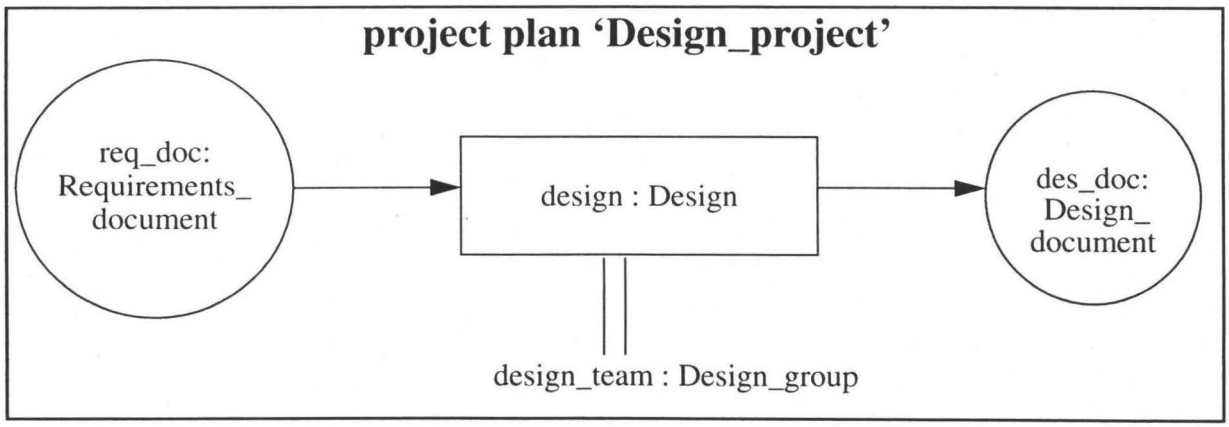

Figure 2: Example Project Plan[1]

4.3 Concepts

In this section we briefly explain the main concepts of MVP-L. A detailed definition of the language follows in the next chapter. The following explanations have an introductory character and should help the novice to understand the basic features of MVP-L.

Firstly, we must distinguish between three domains, which possibly are supported by a process-centered software engineering environment (compare with [9]):

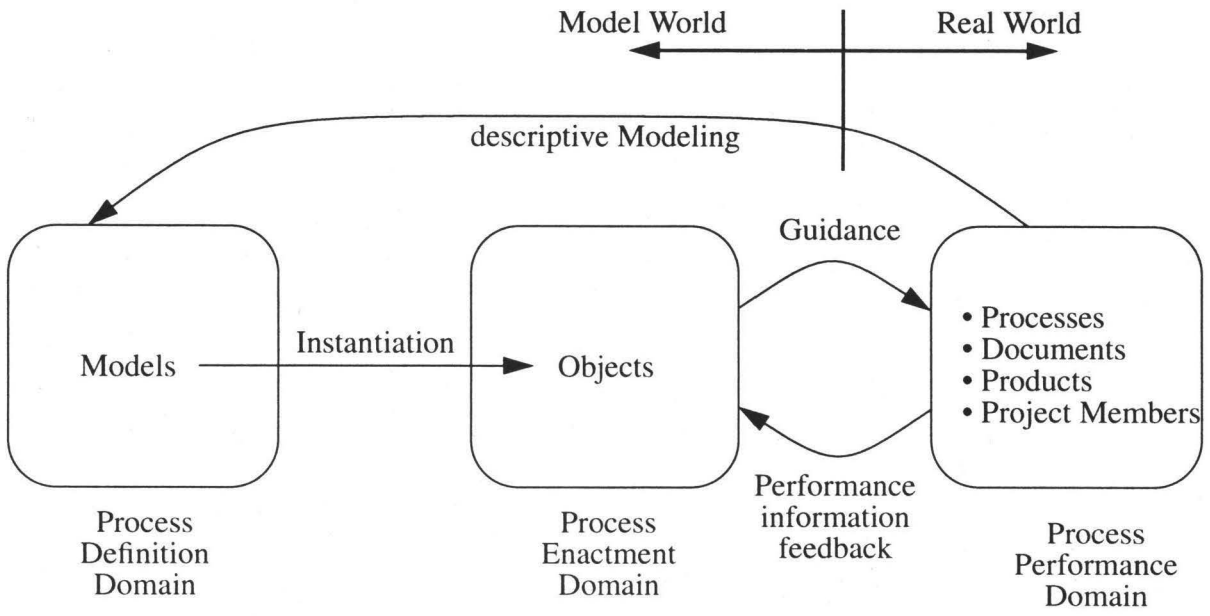

Figure 3: Three Domains of Project Definition, Enactment, and Performance

The process-definition domain is concerned with the identification, description, and analysis of process definitions. Here MVP-L is used for persistent documentation of process knowledge. This domain is part of the experience factory [3].

The process-enactment domain takes the MVP-L descriptions as types to instantiate objects out of the models. Each object has a unique identifier, a defined state, and a predefined behavior. All these object features are described in MVP-L. The process (definition) enactment domain resides within a supporting system, often called process engine, process-centered software engineering environment, or enactment mechanism. This domain is part of the project organization.

Both domains process definition and process enactment deal with representations of real-world objects. Because the representations necessarily leave out details of the objects they represent, this is called the model world.

The process-performance domain is the real world in which agents (humans or machines) are concerned with the development or maintenance of the software, performing processes defined in the project plan. This domain is part of the project organization.

1. In the graphical process model representations, circles represent products, squares represent processes, arrows represent data flow, and double lines represent resource allocations.

The process-enactment domain guides the real-world project by providing information about the project's state, next meaningful steps, and expected qualities of products or processes. The process performance domain has to provide feedback about results, events, or changes in order to allow updating the representation in the computer. There is a cyclic information exchange between both the process-enactment domain and the process-performance domain. Each element of the process performance domain does not necessarily have a corresponding object in the process enactment domain.

In the remainder of this section we introduce the concepts of MVP-L. These concepts affect both the process-definition domain and the process-enactment domain. The process-performance domain and the connection to either of the two other domains is not mentioned here and are discussed elsewhere (for example, see [20]). Throughout the following explanations we use the example in the appendix to explain our ideas.

4.3.1 Elementary Models

Within the MVP-Project we distinguish between a set of elementary types. They reflect our understanding of the basic elements of a software project. In particular, we distinguish between processes, products, and resources.

- Processes are the activities which are performed during a project. They create, read, and modify products.
- The main goal of a process is the development or maintenance of a deliverable software product. In addition to the final product, by-products, artifacts, and parts of a product's documentation are similarly called products. (cf. [7])
- Resources are the entities that are necessary to perform the processes.

Models are used to capture common properties of real-world entities. A model is primarily a description of a real-world phenomenon, focusing on the main features by removing details. Models are described using MVP-L; products, processes, and resources are described by using a **product_model**, **process_model**, or **resource_model**.

In addition to these three basic elements, four kinds of attribute models exist to define observable properties within elementary building blocks. There are three kinds of user-defined attribute models (i.e., **process_attribute_model**, **product_attribute_model**, and **resource_attribute_model**), and one built-in attribute model type (i.e., **global_attribute_model**).

All user-defined models (product, process, resource, and attribute models) written in MVP-L may be seen as type descriptions. They belong to exactly one of the discussed model types. This results in the following type hierarchy (see Figure 4). The model types in the middle of the figure are predefined in MVP-L. The types *model* and *attribute model* on the right side are inserted to illustrate

abstractions of these basic model types. The user-defined models are shown exemplarily on the left side.

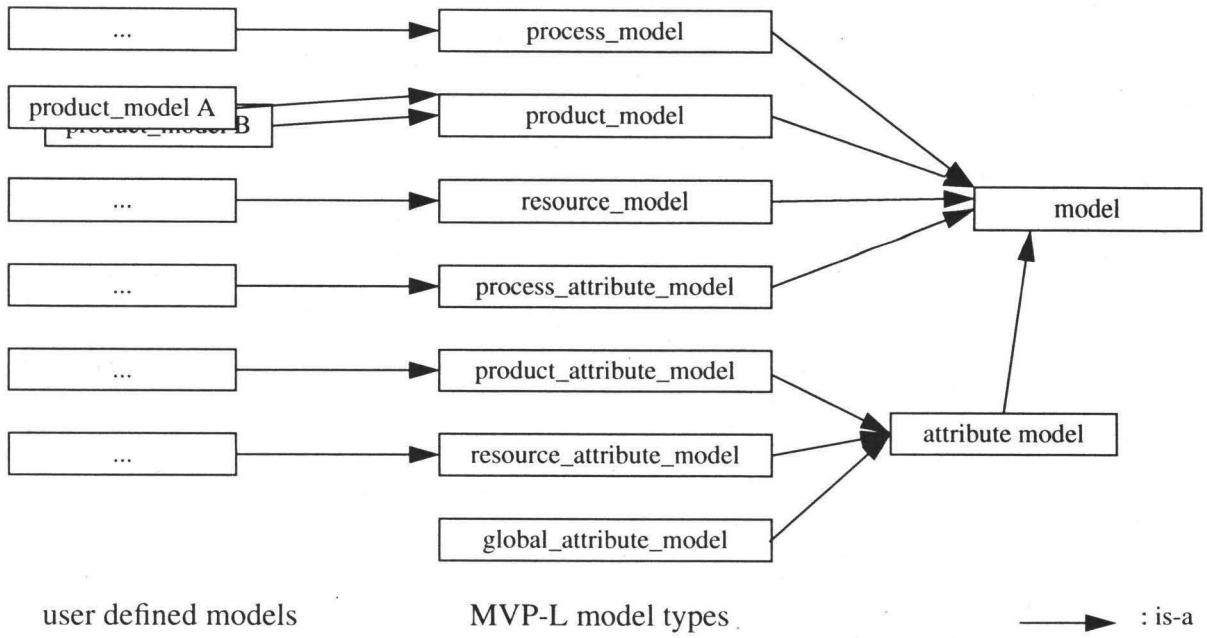

Figure 4: Type Hierarchy of MVP-L Models

One can define process models in MVP-L, which are derived from the first six model types. Only global attribute models are excluded; they are provided by language extensions. A user is not able to specify type hierarchies of his own. Note that an object's type is always the identifier of its model (i.e., A or B in the figure), but the object's model type means its basic MVP model type (i.e., *product_model* in the figure). Instead of saying "object x's model type is *product_model*" we prefer to say "x is a product".

4.3.1.1 Attribute Models

Each attribute model is an elementary one, consisting of a definition of the <attribute_model_type> and the <attribute_manipulation>. The <attribute_model_type> characterizes the type of the values the attributes store, e.g. integer, real, string, boolean, or an enumerated type (see example). The <attribute_manipulation> part will be introduced in Section 5.5.2.5.

Example

```
product_attribute_model Product_status () is
    attribute_type
        ( 'non_existing', 'incomplete', 'complete' );
    ...
end product_attribute_model Product_status
```

Instances of model type <process_attribute_model> can only be declared in a <process_model>. The same is true for <product_attribute_model>s and products, and for <resource_attribute_model>s and resources, respectively.

4.3.1.2 Elementary Product Models

Elementary product models describe the common properties of a class of product objects (or briefly, products). In MVP-L each product is seen as an object consisting of an interface and a body. Information in the <product_interface> is visible to other objects in a project plan. The product attributes are declared in the <exports> clause, and their type must be first imported in the product interface's <import> clause.

Example

```
product_model Requirements_document (status_0 : Product_status) is

    product_interface
        imports
            product_attribute_model PRoduct_status;
        exports
            status : Product_status := status_0;
    end product_interface
    ...
end product_model Requirements_document
```

The product model "Requirements_document" imports the product attribute model "Product_status" of Section 4.3.1.1 to declare an product attribute "status." This attribute can be used by other models. The formal instantiation parameter "status_0" is used to provide the initial value for the attribute.

4.3.1.3 Elementary Resource Models

Elementary resource models describe the active and passive resources used to perform a process. Active resources are organizational entities (e.g., teams) or humans in the real world designated to perform processes. Passive resources are tools which are used to support the performance of a process (e.g., an editor may be used to support a design process). It is important to recognize that traditional software tools can be represented in MVP-L as resources as well as processes. For example, on the one hand, a compiler could be represented as a MVP-L process integrated into a MVP project plan dealing with program development. An editor, on the other hand, may be used within a project plan to support the design process. In this case, the editor would be a passive resource made available to the active resource performing the design process. It is expected that the more we understand the relationships between different processes within project plans and are capable of representing them in more detail, the more we will be able to integrate existing tools as explicit processes rather than as supporting passive resources. Resource models are described in terms of <resource_interface> and <resource_body>. Their meaning is similar to that described for product models. The following example is a fragment of resource model 'Designer' in Figure 2.

Example

```
resource_model Designer(eff_0: Resource_effort) is
    resource_interface
        imports
            resource_attribute_model Resource_effort;
        exports
            effort: Resource_effort := eff_0;
    end resource_interface

    resource_body
        implementation
        -- An instance of this model represents a single member of the design team.
        -- Persons assuming the role of a designer must be qualified.
    end resource_body
end resource_model Designer
```

The instantiation parameter 'eff_0' is used to provide an initial value. In the above example this mechanism is used to state the effort available to this resource for executing some process in the context of a project plan.

4.3.1.4 Elementary Process Models

In the previous two sections we discussed how to represent the elementary products (what is to be developed/maintained in a project) and resources (who does it). Now we discuss the basic MVP-element which combines these basic elements and describes how, in terms of goals, a task has to be performed in a project.

The concept *process* captures the information considered as helpful when an agent assumes a role to perform a task. A process model describes a class of processes. Similar to product and resource models, process models are divided into an interface part and a body. Both of these parts of a process model are more complex than the parts of elementary product or resource models.

Formal Process Interface Parameters

The <process_interface> of elementary process models is described in terms of <imports>, <exports>, <consume_produce>, <context>, and <criteria> clauses, as shown in the example below, which is a part of the model 'Design' of Figure 2. The <process_body> of elementary process models is defined in terms of an <implementation> clause. The <imports> clause lists all externally defined models used to declare formal parameters within the <product_flow> and <context> clauses, or attributes within the <exports> clause. The <exports> clause lists all externally visible attributes which can be used by other models.

Example

```
process_interface
    imports
        process_attribute_model Process_effort;
        product_model Requirements_document, Design_document;
    exports
        effort: Process_effort := eff_0;
```

```
        max_effort: Process_effort := max_effort_0;
    product_flow
        consume
            req_doc: Requirements_document;
        produce
            des_doc: Design_document;
        consume_produce
    context
    entry_exit_criteria
        local_entry_criteria
            (req_doc.status = 'complete') and (des_doc.status = 'non_existent'
                or des_doc.status = 'incomplete');
        global_entry_criteria
        local_invariant
            effort <= max_effort;
        global_invariant
        local_exit_criteria
            des_doc.status = 'complete';
        global_exit_criteria
end process_interface
```

The <exports> clause lists all externally visible attributes which can be used by other models. In process model 'Design', attributes 'status' and 'effort' of types 'Process_status' and 'Process_effort', respectively, are made available to all models importing process model 'Design'.

The <product_flow> clause lists as formal parameters the products consumed, produced or both. These formal parameters are typed in terms of imported product models. They define the local process interface. In the case of process model 'Design', a product 'req_doc' of type 'Requirements_document' is consumed and a product 'des_doc' of type 'Design_document' is produced.

The <context> clause lists as formal parameters all objects other than the ones listed in the <product_flow> clause which are necessary to define the <process_interface>. These global interface parameters are also typed in terms of imported models. They define the global process interface. The example process model 'Design' does not contain any global objects.

The <criteria> clause contains criteria necessary to enact the process, criteria for valid states during enactment, and criteria expected upon enactment completion. Criteria are specified as boolean expressions. The expression following the keyword **local_entry_criteria** defines the criteria necessary to execute the process in terms of locally defined attributes and local interface parameters. The expression following the keyword **global_entry_criteria** defines the criteria necessary to execute the process in terms of global interface parameters (i.e., the ones defined in the <context> clause of the process).

In this example, the local invariant specifies that the actual effort spent for any instance of the process model 'Design' should never exceed a value specified by 'max_effort'. No global invariant exists in the example.

In the above example, the local entry criteria state that any process of type 'Design' can only be executed if attribute 'status' of product 'req_doc' has value 'complete' and attribute 'status' of

product 'des_doc' has either value 'non_existing' or 'incomplete'. No global entry criteria exist. The expression following the keyword **local_entry_criteria** defines the criteria expected upon completion of process execution in terms of local attributes and the local interface. The expression following the keyword **global_exit_criteria** defines the criteria expected upon completion of process execution in terms of the global interface. In the above example, the locally expected results upon completion are that attribute 'status' of product 'des_doc' has value 'complete'. No global exit criteria exist in this case.

The <implementation> clause describes how an elementary process is to be performed. This can either be a call of a tool, or simply an informal comment characterizing the task at hand for performance by a human.

Processes are related to products via explicit <product_flow> relationships, to attributes via <criteria> clauses, and to resources via a separate <process_resources> clause. In the example of process model 'Design', a resource 'design_team' of type 'Design_group' is designated to execute any process of type 'Design'. This is not shown in the above example, but is specified in the corresponding process model of Appendix A.

Instrumentation

Instrumentation of a process refers to the explicit definition of data to be collected during process enactment. The use of the data is assisted by some quality model (e.g., fault detection rate of a particular component test process). In general, a quality model (QM) can be seen as a function accepting some measurement data and producing a statement about whether a particular quality goal has been achieved or not. The GQM approach may be used to specify the interpretation rules for quality models in a structured way [2].

Processes can be related to quality models via attributes. Any quality model $QM=f(m_1,m_2,....,m_n)$, defined in terms of n measures m_i, can be reflected via attributes in MVP-L. Each process measure m_i is defined as an attribute within a process model. Similarly, product and resource measures are defined as attributes within product and resource models, respectively. Attributes represent actual values which are compared to target values. Target values are the to-be-achieved values defined during planning. They may be represented by invariants or criteria or may exist outside an MVP-L model. Actual values are collected during enactment automatically (e.g., by binding this attribute to some data collection tool) or manually (e.g., by asking the human agent). In our example process 'Design', attributes 'effort' of user-defined type 'Process_effort' and 'status' of built-in type 'Process_status' (with values disabled, passive, or active) are defined. The value 'eff_0' is used as a target value not to be exceeded by any execution of a process of type 'Design'. A complete description of the attribute models 'Process_effort' and 'Process_status' is included in Appendix A. The <attribute_manipulation> parts consist of state transitions specifying what kinds of events cause what change of an attribute value. The <attribute_manipulation> of user-defined attributes have to be provided by the user; the <attribute_manipulation> of built-in types is pre-defined. The <attribute_manipulation> part of attribute 'Process_status' reflects the idea of executing processes as inherently parallel entities only constrained by their entry and exit criteria. A process has status 'disabled' if its entry criteria evaluate to false; status 'enabled' if its entry criteria evaluate to true but it has not been invoked for execution; and status 'active' if its entry criteria evaluate to true and it has been invoked for execution. We distinguish between two kinds of triggers: invocations and events. The user is able to release the events 'start' and 'com-

plete' which are received by particular processes, i.e. the ones to be started or completed. Events such as 'the entry criteria become true' or 'the entry criteria become false' are typically the result of changes outside the process in question.

4.3.2 Project Plans

This section discusses the MVP-L representation of project plans with the emphasis on the instantiation of process objects within a project plan and process enactment. Throughout this section, fragments of project plan 'Design_project_2' from Appendix A are used for illustration purposes.

4.3.2.1 Instantiation of Processes

Software process models are instantiated in the context of a <project plan>. A <project_plan> is described in terms of <imports>, <objects>, and <plan_object_relations> clauses. The imports clause lists all models used to declare the process, product, and resource objects that make up the project plan. The objects are declared in the <objects> clause. The objects are interconnected according to their formal interfaces in the <plan_object_relations> clause. Type checking can be performed at compile time. A project plan is interpreted by a process engine in order to enact the processes. This enactment machine may be a human or a computer.

The following example project plan is used to demonstrate all three project plan clauses:

Example

```
project_plan Design_project_2 is
    imports
        product_model Requirements_document, Design_document;
        process_model Design;
        resource_model Design_group;
    objects
        requirememements_doc: Requirements_document('complete');
        design_doc: Design_document('non_existent');
        design: Design(0, 2000);
        design_team: Design_group(0);
    object_relations
        design(req_doc => requirements_doc, des_doc => design_doc,
        designers => design_team);
end project_plan Design_project_2
```

This project plan does not require much explanation, because it is equivalent to the graphical depiction of the project plan in Figure 2 of Section 4.1. It consists of four objects: one process 'design', two products 'requirements_doc' and 'design_doc', and one resource 'design_team'. The interconnection of these products and the resource to the only process 'design' are performed in conformance with the formal interface specification of process model 'Design'. We assume here that a complete requirements document with name 'requirements_doc' is provided, that no design document 'design_doc' exists at this point in time, and that the maximum time allowed for the performance of process 'design' is 2000 time units. Only members of the 'Design_group' are allowed to perform process 'design'.

4.3.2.2 Enactment Model

The enactment model in MVP-L is based on the notion of project state. A project state is defined as the set of all attribute values (i.e., all attributes of all objects instantiated within a project plan). The initial project state is defined in terms of the initial values of all user-defined attributes, and the derived values of built-in attributes. The values of attributes of built-in types 'Global_time' and 'Global_date' are externally defined. The values of attributes of built-in type 'Process_status' depend on the entry and exit criteria. The only triggers to change the project state are user invocations of the kind 'start(<object_id>)' and 'complete(<object_id>)' to start and complete processes, or the invocation 'set(...)' to reflect external changes of attributes. In each case, the new values of all user-defined and built-in attributes are computed to determine the new project state. This new project state indicates which processes are in execution (i.e., value of process status is 'active'), ready for execution (i.e., value of process status is 'enabled'), or not ready for execution (i.e., value of process status is 'disabled').

4.3.3 Complex Process Models

This section discusses the MVP-L representation of complex process models with the emphasis on the separation of interface from body and other modularization concepts. Throughout this section, fragments of the refinement part of process model 'Design' from Appendix A are used for illustration purposes.

4.3.3.1 Modularization Concepts

Elementary process models can be aggregated to create more complex process models. The same is true for any other kind of model. Each object is exactly aggregated once (i.e., the aggregation hierarchy builds a tree). The most abstract product and processes are aggregated by the project plan. On the other hand, complex models can be refined into more elementary models to add more detail.

Refinements in MVP-L are level complete. That means that the refined level entirely replaces the parent level. A refinement of a process model is described in the <refinement> clause of the <process_body> part. The <refinement> part consists of <imports>, <objects>, <object_relations>, <interface_refinement>, <interface_relations>, and <attribute_mapping> clauses.

The model types used and objects declared at the refined level are listed in <imports> and <objects> clauses. In our example, process model 'Design' is refined into the two elementary processes 'hld' and 'lld' of types 'High_level_design' and 'Low_level_design'; and two elementary products 'hl_des_doc' and 'll_des_doc'.

Example

```
refinement
    imports
        product_model High_level_design_document, Low_level_design_document;
        process_model High_level_design, Low_level_design;
```

```
objects
    hl_des_doc: High_level_design_document('non_existent');
    ll_des_doc: Low_level_design_document('non_existent');
    hld: High_level_design(0, max_effort / 2);
    lld: Low_level_design(0, max_effort / 2);
```

It is important to note that typically a process refinement also requires a refinement of the process interface. In the above case, the refinement of 'Design' into 'hld' and 'lld' requires also a refinement of the consumed product 'des_doc' into 'hl_des_doc' and 'll_des_doc'. These two elementary products are not objects defined at the refinement level, but formal parameters refining the formal parameter 'des_doc' at the parent level. This is reflected by listing the actual refinement of the parent interface parameter in the <interface_refinement> clause:

Example

```
interface_refinement
    des_doc = ( hl_des_doc & ll_des_doc );
```

The <interface_relations> clause describes connections between the interface parameters of each process object at the refined level and the interface parameters at the parent level. MVP-L uses an Ada-like notation to indicate these connections for each process at the refinement level. In the case of process model 'Design', the interface parameters 'req_doc', 'hl_des_doc', and 'll_des_doc' are satisfied as indicated below:

Example

```
process_interface
...
product_flow
    consume
        req_doc: Requirements_document;
    produce
        des_doc: Design_document
    consume_produce
...
process_body
    refinement
    ...
        interface_refinement
            des_doc = ( hl_des_doc & ll_des_doc );
        interface_relations
            hld(hl_des_doc => hl_des_doc, req_doc => req_doc);
            lld(ll_des_doc => ll_des_doc, hl_des_doc => hl_des_doc);
    ...
```

In this example, the formal consume and produce parameters 'req_doc' and 'hl_des_doc' of process 'hld' are bound to the formal interface parameters 'req_doc' and 'b21' at the parent level. Similarly, the formal parameters of process 'a2' are bound to the formal interface parameters at the parent level. The interface connections are consistent in their types as can be seen from the

definitions in product models 'Requirements_document' and 'Design_document', and process model 'Design'.

5 MVP-L Language Definition

The previous chapter introduced the basic concepts of MVP-L. In this chapter we provide a detailed definition of the software process modeling language. The explanation is syntax-oriented. Each part of an MVP-L description, which is a set of project plans and models, is explained separately, moving from the bottom (meaning the elementary language constructs) to the top. This section is not expected to be a document for teaching MVP-L. It should serve the user of MVP-L as a reference manual.

5.1 Notation of Syntax

In this chapter we will use the following constructs to define the syntactical rules of MVP-L:

<non-terminal>	Non-terminals begin with a left angle bracket and terminate with a right angle bracket.
::=	Separates the name of a non-terminal and its production rule.
[]	Parts of the rule enclosed in brackets may be ignored.
{ }	Parts of the rule enclosed in braces may be ignored or repeated any number of times.
{}+	A '+' symbol appearing directly after a closing brace indicates that the part of the rule enclosed in the braces must appear at least once.
{.../// "*string*"}	Repetitions of the symbols appearing before a double slash are separated by the string enclosed in quotation marks.
|	Separates two alternatives.
()	Parentheses group symbols.
"*symbol*" or '*symbol*'	Quotation marks enclose terminal symbols.
bold	Terminal symbols not enclosed in quotation marks are printed bold.
.	Terminates a production rule.

5.2 Vocabulary and Representation of Terminal Symbols

Terminal symbols of MVP-L are represented using bold characters. The language distinguishes between upper case and lower case characters. Spaces, tabs and newline characters separate terminal symbols, except when they appear in comments.

The following rules describe the permissible terminal symbols in MVP-L:

1. Numbers

Numbers have integer or real values and optionally a symbol indicating a negative or positive sign. They are expressed using the decimal system.

<integer> ::= ["-"] { digit }+ .

<real> ::= ["-"] { digit }+ "." { digit }+ .

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .

There are no ranges or boundaries on numbers provided or required by this language definition, so one should be careful when porting a MVP-L description from one enactment system to another one. But it is assumed that at least the standard ranges provided by runtime systems (e.g., 32-bit representation of integers) are available.

2. Strings

Strings are concatenations of single characters enclosed in quotation marks ("). Because a quotation mark itself is not an element of the set of characters, it cannot appear in a string. An empty string is represented by "".

<string> ::= '"' { character } '"' .

<character> ::= <digit> | <alpha_char> | <ident_char> | <special_char> .

<alpha_char> ::= "A" | ... | "Z" | "a" | ... | "z" .

<ident_char> ::= "_" | "-" .

<special_char> ::= "!" | "@" | "#" | "$" | "%" | "^" | "&" | "*" | "(" | ")" | "+" | "=" | "{" | "}" | "[" | "]" | ":" | ";" | "'" | "~" | "<" | ">" | "." | "," | "?" | "/" | "|" | "\" | "`" | " " .

Note, that using the character '-' is ambiguous. For example, 'a-b' is an identifier, whereas 'a - b' is an expression.

3. Identifiers

Identifiers begin with a letter and may be followed by letters, digits, underscores, or minus signs.

<object_id> ::= <ident> .

<process_object_id> ::= <ident> .

<model_id> ::= <ident> .

<object_model_id> ::= <ident> .

<project_id> ::= <ident> .

<process_model_id> ::= <ident> .

<product_model_id> ::= <ident> .

<resource_model_id> ::= <ident> .

<attr_model_id> ::= <ident> .

<event_id> ::= <ident> .

<invoc_id> ::= <ident> .

<object_of_a_process> ::= <ident> .

<object_of_the_plan> ::= <ident> .

<tool_identifier> ::= <ident> .

<ident> ::= <alpha_char> { <alpha_char> | <ident_char> | <digit> } .

The different non-terminals were introduced for the sake of readability and to represent semantic rules in the syntax (e.g., a process object's name must always be a <process_object_id>).

4. Symbols

Symbols are user-defined values and were introduced to increase the readability of process descriptions. They are similar to enumeration types of other imperative programming languages (cf. Modula-2 [33]), but no defined relation exists between two symbols. Only tests for equality and inequality are allowed. Symbols start and end with an apostrophe.

<symbol> ::= "'" <ident> "'" .

5. Name Constants

Name constants differ from other terminal symbols in that way that they alias a special value. In MVP-L the only name constants used are these for the boolean values *true* and *false*.

<boolean> ::= **true** | **false** .

6. Operators and Key Words

Operators and key words of MVP-L are listed below (see 5.7 "Key Words" on page 45). Key words cannot be used as identifiers, and because identifiers must start with an <alpha_char>, operators cannot be misinterpreted as user defined names.

7. Comments

Comments may appear in a MVP-L process description only in selected places, which are specified in the syntax. This was chosen to enable the derivation of the original MVP-L text from an intermediate representation. There are two ways of inserting comments into MVP-L process descriptions. They are either enclosed in braces or start with a double minus sign and terminate with end of line.

In the first case, every character may appear in between, except a right brace. Also spaces, tabs and newlines are allowed. Comments should not be used to separate symbols in a single line. This form of representation was chosen because comments in MVP-L descriptions are usually not limited to a single line.

For short annotations one can use the second alternative. This kind of comment is terminated by a newline character.

<comment> ::= "{" { <character> } "}" | "--" { <character> } .

5.3 Declaration and Scope

Because models are the only concept which can reference other objects and which cannot be declared recursively, an identifier (of a model or of an object) is visible either in a single model or in the whole project description. Identifiers are unique in the sense that they may not be declared twice in the same scope.

5.3.1 Declaration and Scope of Model Types

1. Model Type Definition

To express all relevant aspects of a real world project, seven kinds of models are provided. These model types are used by the MVP-L user[1] to describe real-world software development projects. The model types are described elsewhere as listed below.

Process Model (see 5.5.1.3 "Process Models" on page 37)

Product Model (see 5.5.1.4 "Product Models" on page 38)

Resource Model (see 5.5.1.5 "Resource Models" on page 38)

Process Attribute Model (see 5.5.2.1 "Process Attribute Models" on page 40)

Product Attribute Model (see 5.5.2.2 "Product Attribute Models" on page 40)

Resource Attribute Model (see 5.5.2.3 "Resource Attribute Models" on page 41)

Global Attribute Model (see 5.5.2.4 "Global Attribute Models" on page 41)

2. Model Type Import

Model type identifiers are visible in the whole project definition. However, to use these types in other model definitions the types' identifiers have to be imported. This feature was introduced to increase the visibility of relationships between project elements.

<imports> ::= **imports** { <library_part> ";" } .

<library_part> ::= <model_type> { <model_id> // "," }+ .

Table 2 expresses what MVP-L elements may import what element types.

Note: Attribute models do not have an import clause. Because there is no need to import instantiation parameter types, there are no types to be imported. The corresponding lines were removed from Table 2.

5.3.2 Declaration of Attributes

1. Visibility of Attributes

Attributes describe the state of a model instance and collectively of the whole project. To use these elements in other model instances, the attributes have to be made visible explicitly. There-

1. A user of MVP-L typically will be an environment builder of process-centered software engineering environments. Project guidance or examinations of existing process models by project members should be done using graphical user interfaces instead of raw textual representations.

Model Type	is allowed to import model type						
	Process Model	Product Model	Resource Model	Process Attribute Model	Product Attribute Model	Resource Attribute Model	Global Attribute Model
Process Model	●	●	●	●			●
Product Model		●			●		●
Resource Model			●			●	●

Table 2: Import Relations Allowed between Model Types

fore, the export clause exists to allow access by another model instance. Compare this to *public* declarations in Ada. Nevertheless, the declaration of private or local attributes is not possible in MVP-L.

<exports> ::= **exports** { <attribute> ";" } .

Additionally, the identifier of an object's formal parameter (see <instantiation_parameter> clause on page 38) is visible to other models (i.e., they must use the formal parameter to pass initial values for instantiation to the object).

2. Attribute Declarations

MVP-L is a strongly typed language. Therefore, all attribute values have to be declared. Every attribute has a name which is unique in the scope it is declared in (i.e., process model, product model, resource model, or project plan). An attribute's type is either a simple type or an attribute model (Section 5.5.2). It should be noted that in process declarations only simple types (i.e., <attribute type>) and process attribute models are allowed to specify the type of an attribute. Correspondingly the use of product attribute models (page 40) and resource attribute models (page 41) is limited to product or resource models, respectively.

<attribute> ::= <object_id> ":"
(<attribute_type> | <attr_model_id>)
[":=" <attribute_value>] .

<attribute_type> ::= **integer** | **real** | **string** | **boolean** .

<attribute_value> ::= <expression> .

5.3.3 Inclusion of Objects

During software development or maintenance projects, different elements are needed to produce or change a software product. Usually they do not become part of an instantiated process itself in the sense of a building block in the way that attributes do. Moreover, several elements (e.g., tools, person performing a process) are associated with the process model in order to support perform-

ance of the task. Mostly they represent elements shared between several processes and thus their use has to be planned and scheduled.

These objects also do not become part of the software product. They are only used during development or maintenance to support the corresponding tasks.

It should be noted that neither personnel nor tools are passed from an aggregate process to a subprocess (see Section 5.5.1.2.). The only location where resources may be instantiated is in a project plan (unlike the instantiation of products or processes which is specified in aggregating processes). Therefore the <objects> clause of <personnel_assignment> and <tool_assignment> clauses may not contain instantiation parameters.

1. Inclusion of Personnel

Most activities during a software project cannot be done automatically. Human beings are part of the mechanism for project enactment to solve difficult, complex, and intuitively understood problems. Therefore, we provide language constructs for explicitly describing the staff who are involved in a software process.

Firstly the types describing different kinds of people or groups in a project (i.e., resource models) have to be imported, and secondly the instances of every type that are needed to support the associated task must be specified.

<personnel_assignment> ::= **personnel_assignment**
[<imports> <objects>] .

2. Inclusion of Tools

Similarly to personnel assignments, the tool types (i.e., resource models) have to be imported and the instantiation of objects representing tools has to be declared.

<tool_assignment> ::= **tool_assignment** [<imports> <objects>] .

3. Inclusion of Other Objects

Not all objects can be categorized either as personnel or as tools. During a project's lifetime other kinds of objects are used, for example product documentation standards, results of former process instantiations, or programming guidelines not supported by tools. We call all of these objects a process's context.

In a context clause the instances of a process's context are specified. Types for objects specified in a <context> clause have to be imported. Their basic model type has to be resource_model. For example, programming guidelines should be specified in the <context> clause.

<context> ::= **context** { <object_decl> } .

4. Object Declarations

To provide information for storage management and analysis of the project description, not only must the types needed to define a model be specified, but the instances of each type must also be given. This is called an object definition. The object identifiers are visible in the model they are defined in. Additionally, a type has to be specified for every object. Optionally instantiation parameters are provided. These actual parameters must be compatible with the formal parameters

specified in the model's appearing type definition (see <instantiation_parameters> on page 38). This specification of initial values is permitted only in bodies of project plans and process models. Every object declaration terminates with a semicolon.

<objects> ::= **objects** { <object_decl> } .

<object_decl> ::= <object_id_list> ":" <type_decl> ";" .

<type_decl> ::= <model_id> ["(" <parameter_list> ")"] .

<parameter_list> ::= { <attribute_value> // "," }+ .

<object_id_list> ::= { <object_id> // "," }+ .

5.4 Expressions

Expressions are rules to compute a single value. They can appear where a single value is expected. Operands refering to atomic values (variables and constants in procedural programming languages) are combined using operators.

5.4.1 Operands

Operands refer to a single value with a special type.

Objects, i.e., instances of process, product, or resource models, are used as operands by referencing the object identifiers declared in the <object_decl> of the model containing the references. Attribute values can be used as operands in the same way. Symbols, i.e., user-defined values of attribute models, start and end with an apostrophe. Any textual constant (e.g., "string", 1234, **true**) may appear as an operand. Finally, tool invocations are also considered operands because they return a value.

<operand> ::= <object_value> | <symbol> | <integer> | <real> | <string> | <boolean> | <tool_invocation> .

<operand2> ::= <symbol> | <integer> | <real> | <string> | <boolean> .

<object_value> ::= <object_id> ["." <object_id>] .

The <operand2> clause was introduced for use in attribute mappings.

Tools declared as resources can be invoked by using them as an operand. Logically they can be used everywhere an expression is expected. Invocations start with the reserved word **call**. The first identifier in the parentheses enclosing the arguments must be the tool's name, eventually followed by a list of actual parameters, which is passed to the tool's interface. In previous versions of MVP-L, the return value's type of a tool invocation was not specified. When implementing the first prototype of MVP-S its type was assumed to be real per default. To allow more return value types, one may specify intervals or enumerations after the call. This is done by specifying an interval of numbers (**real** or **integer**) or an enumeration of discrete values after the keyword **in**. The resulting type specification may only use constant values. For reasons of compatibility with former versions of MVP-L, the return value's type is optional. Default type is **real**.

<tool_invocation> ::= **call** "(" <tool_identifier>
["(" { <expression> // "," }+ ")"] ")"
[**in** (<integer_interval> | <real_interval> |
<string_enumeration> | <bool_enumeration> |
<symbol_enumeration>)] .

<tool_identifier> ::= <resource_model_id> .

<integer_interval> ::= "[" <integer> ".." <integer> "]" .

<real_interval> ::= "[" <real> ".." <real> "]" .

<string_enumeration> ::= "[" { <string> // "," }+ "]" .

<bool_enumeration> ::= "[" { <boolean> // "," }+ "]" .

<symbol_enumeration> ::= "[" { <symbol> // "," }+ "]" .

5.4.2 Operators

Operators are used to combine operands or expressions to form more complex expressions. In MVP-L we distinguish between several classes of operators with different precedence. Unlike previous versions of the MVP-L grammar, the operator precedence is now represented in the syntax. The operators are listed in Table 3 have from the highest to the lowest binding strength.

Precedence of Operators	
-, not (unary)	high
*, /	
+, - (binary)	
=, !=, <, <=, >, >=	
and	
or, xor	low

Table 3: Precedence of Operators

<expression> ::= <arith_exp> <compare_op> <arith_exp> |
<arith_exp> .

<compare_op> ::= "=" | "!=" | "<" | "<=" | ">" | ">=" .

<arith_exp> ::= <arith_exp> "+" <arith_term> |
<arith_exp> "-" <arith_term> |
<arith_exp> **or** <arith_term> |
<arith_exp> **xor** <arith_term> |
"-" <arith_term> |
<arith_term> .

<arith_term> ::= <arith_term> "*" <arith_factor> |
<arith_term> "/" <arith_factor> |
<arith_term> **and** <arith_factor> |
<arith_factor> .

<arith_factor> ::= <operand> | "(" <expression> ")" |
not <arith_factor> .

The symbol "-" can appear both as a monadic operator expressing the sign of a number or as a dyadic operator expressing subtraction, but its appearance in a process description is unambiguous. All operators require at least one operand. Operators may be applied only to operands of dedicated types. We give here for each operator its signature (see Table 4).

Operator	Permissible Argument Type		Result Type
	First	Second	
=, !=, <, <=, >, >=	Real	Real	Boolean
	Real	Integer	
	Integer	Real	
	Integer	Integer	
	Boolean	Boolean	
=, !=	String	String	
	Symbol	Symbol	
+, - (monadic)	Real		Real
	Integer		Integer
+,-, *	Real	Real	Real
	Real	Integer	Real
	Integer	Real	Real
	Integer	Integer	Integer
/	Real	Real	Real
	Real	Integer	Real
	Integer	Real	Real
	Integer	Integer	Real[a]
and, or, xor	Boolean	Boolean	Boolean
not	Boolean		Boolean

Table 4: Signatures of MVP-L Operators

a. Please note that '/' is the division operator. No DIV operator with result type integer exists in MVP-L.

5.4.3 Requests

Requests are special expressions which instruct the process engine (enactment machine) to request input from a user. Evaluation of the request expression initiates a communication with the user (e.g., result of a review).

<mail_request> ::= **request** "(" <message_string> ")" .

<message_string> ::= <string> .

The sole argument of a request is a string which is presented to the user in the communication that prompts for a value.

5.5 Models

The choice of MVP-L language features was driven by an empirical understanding of the concepts used by application experts (e.g., developers, maintainers, managers, and other project personnel) to describe and talk about software development and maintenance processes [12], [28]. Frequently used application concepts referred to 'processes of some type', 'products of some type', 'humans or organizational entities in charge of some process', 'tools', 'the use of some product as input to a process', 'some product produced as the result of a process', 'the fact that a process can only be started if certain properties are fulfilled', and 'the fact that upon completion of a process certain properties must be fulfilled'. The mentioned properties typically refer to attributes of processes and/or products according to some quality model. In MVP-L, we chose language features to reflect these application concepts in a natural way:

<model_type> ::= **process_model** | **product_model** |
resource_model | **global_attribute_model** |
process_attribute_model |
product_attribute_model |
resource_attribute_model .

Process, product, and resource models were chosen to represent the types of objects referred to by the application experts. Resources include humans, organizational entities, and tools. Attribute models were chosen to represent the characteristics used to describe the properties related to the beginning and completion of processes. The functionality of processes is described in terms of the data flow relations 'consume' and 'produce'. The behavior of processes is described in terms of entry and exit criteria. On the one hand, these language features enable the formal representation of implicit project knowledge. The benefits of formal representations include unambiguity and analyzability. On the other hand, the one-to-one mapping of application concepts onto elements of the MVP-L representation results in natural models. The benefits of natural models include the ability for application experts to recognize their applications in the formal models and thereby validate them.

In the following we first introduce the general kinds of objects identified for modeling a software project. Secondly, we show how we express software process quality models in terms of attributes related to objects or global attributes associated with the whole project.

5.5.1 Object Models

The representation of a real-world software project is not a comprehensive, single piece of strongly coupled information. There are many types of different elements correlated in multiple ways. Their instances form a heterogeneous network of interrelated data, namely the software process representation. Therefore, MVP-L provides mechanisms for specifying the elements' interface and their behavior, for classifying them, and for defining their data in terms of quantifiable expressions.

5.5.1.1 Model Interface

A common problem in most software projects is the inability to recognize ad hoc relationships and communication channels between objects. Because MVP-L aims at an overall improvement of software processes, this fact influenced the formalism's design. Therefore, most of this effort went into process model interfaces.

1. Process Interface

A process's interface is described using four parts, which are enclosed by the keywords **process_interface** and **end process_interface**.

<process_interface> ::= **process_interface** [< comment>]
<import_export>
<product_flow> <context> <criteria>
end process_interface .

The <import_export> clause specifies the use of other types in this process model and the visibility of attributes for use in other models, respectively.

<import_export> ::= <imports> <exports> .

The product flow between process models is represented by information specified in <product_flow> clauses. One can identify two meanings of this specification, a static one and a dynamic one:

> Analyzing only the project description (i.e., static text), the clause specifies in an abstract fashion how this process affects the status of processes. One should not be mislead by the term *product flow*. A product is not really consumed by this process, i.e. there is no destruction of the element. We only describe what objects represent the input of a process and which types they have.

Correspondingly, the <produce> clause expresses that during process enactment events are generated to signal the creation of a new object representing a product component. When an object is modified during process enactment, the <consume_produce> clause has to be used. This expresses that a new version of the specified object is generated.
When a product is produced, consumed, or consume_produce'd, a corresponding event is sent to the product; **start.produce**, **start.consume**, or **start.consume_produce** at the beginning of the process, and **complete.produce**, **complete.consume**, **or complete.consume_produce** at its termination (see <process_rel> clause on page 42). In the <selection> clause one can specify the names of the attributes which should receive these events; the attributes listed in the <selection> clause have to appear in the corresponding product model's <export> clause.

```
<product_flow>          ::=  product_flow
                             [ <comment> ] <consume>
                             [ <comment> ] <produce>
                             [ <comment> ] <consume_produce> .

<consume>               ::=  consume
                             { <object_id> ":" <product_model_id>
                             [ < selection> ] ";" } .

<produce>               ::=  produce
                             { <object_id> ":" <product_model_id>
                             [ < selection> ] ";" } .

<consume_produce>       ::=  consume_produce
                             { <object_id> ":" <product_model_id>
                             [ < selection> ] ";" } .

<selection>             ::=  "(" { <ident> // "," }+ ")" .
```

The third specification in a process interface, namely the <context> clause, describes the usage of objects other than resource model instances or product model instances. It is described in Section 5.3.3.

Based on our current knowledge about the behavior of software processes, we cannot derive an algorithmic expression for the whole task of a real-world software development or maintenance task. The order of execution of the independent, encapsulated processes is determined by the evaluation of rules.[1] The <criteria> clause contains criteria necessary to start the process, criteria to characterize enactment according to plan, and criteria expected upon execution completion. The type of the expression in the <criteria> clause must be of type boolean. The <criteria_expression> following the keyword **local_entry_criteria** defines the criteria necessary to execute the process in terms of locally defined attributes and local interface parameters. The <criteria_expression> following the keyword **global_entry_criteria** defines the criteria necessary to execute the process in terms of global interface parameters . Invariant criteria should remain true over the process lifetime; invalidation of the expression causes a signal to be sent to the user. It is left open to the implementation of the enactment mechanism whether interpretation of the project plan is resumed or not. The appearance of the keywords **local_entry_criteria**, **global_entry_criteria**, **local_invariant**, **global_invariant**, **local_exit_criteria**, or **global_exit_criteria** is followed by a single boolean expression; if no criteria is specified the keyword must appear.

The difference between local and global criteria is made with respect to the use of attributes. In a process model's local entry criteria, local invariant, and local exit criteria only attributes of the process itself and attributes of products appearing in the <product_flow> clause may be used. In a process model's global entry criteria, global invariant, and global exit criteria additionally attributes of objects defined in the model's <context> clause may appear. When a global criteria is specified without an attribute of an object defined in the context, this is a fault.

---

1. For a detailed discussion of rule-based software process modeling see [14].

<criteria> ::= **entry_exit_criteria**
[<comment>] <entry_criteria>
[<comment>] <invariant>
[<comment>] <exit_criteria> .

<entry_criteria> ::= **local_entry_criteria** [<criteria_expression>]
global_entry_criteria
[<criteria_expression>] .

<invariant> ::= **local_invariant** [<criteria_expression>]
global_invariant [<criteria_expression>] .

<exit_criteria> ::= **local_exit_criteria** [<criteria_expression>]
global_exit_criteria [<criteria_expression>] .

<criteria_expression> ::= <expression> ";" .

2. Product Interface

Because a product is a passive object, only other product models, product attribute models, and global attribute models can be imported (see Table 2: "Import Relations Allowed between Model Types" on page 24). The variety of relations to other objects, like in process declarations, are not supported.

<product_interface> ::= **product_interface**
[< comment>]
<import_export>
end product_interface .

3. Resource Interface

The resource interface is identical to the product interface (see Table 2: "Import Relations Allowed between Model Types" on page 24).

<resource_interface> ::= **resource_interface**
[< comment>]
<import_export>
end resource_interface .

5.5.1.2 Model Body

In the second part of the model specification we have to explain what the types and objects specified in the interface part are needed for and how they are used. This is called the body of a model. There are two types of bodies appearing in models, namely elementary and refined ones.

<process_body> ::= **process_body**
([<comment>] <process_refinement> |
<process_implementation> [<comment>])
end process_body .

<product_body> ::= **product_body**
([<comment>] <product_refinement> |
<product_implementation> [<comment>])
end product_body .

<resource_body> ::= **resource_body**
([<comment>] <resource_refinement> |
<resource_implementation> [<comment>])
end resource_body .

1. Elementary Model Body

An elementary body of a model expresses the atomicity of this element for understanding, planning, enactment, reuse, or analysis. A project element containing an elementary body (also called an *elementary project object*) is treated as a single element, similar to transactions in database systems, files in network file systems, or messages in electronic mail systems. Elementary project objects are the smallest object of management in a MVP system. Nevertheless, a user of the MVP system must be able to manage smaller units (e.g., changing a requirement in a requirements document which is represented in this case as an elementary product in MVP-L).

For elementary process elements, only the interface of this model needs to be specified. Its behavior is left open and depends on the implementation. MVP-L does not provide language constructs for specifying functionality of elementary processes or detailed representations of products and resources. Thus, information about these aspects has to be documented using comments, and no support can be provided by a MVP-L runtime system in monitoring the expected enactment in a project.

<process_implementation>::= **implementation** .

<product_implementation> ::= **implementation** .

<resource_implementation>::= **implementation** .

2. Complex Model Body

The representation of information on multiple levels of abstraction enables the use of representations of a system or system's components for different purposes. In MVP-L one is free to decide at which level of abstraction to describe project elements. Even the description of one single piece of information on different levels is allowed. Therefore, some support has to be provided for maintaining the consistency of different interface declarations and structure definitions.

Elementary models can be aggregated to create more complex models in a bottom-up manner to build more abstract representations. Complex models can be refined into more elementary models in a top-down manner to add more detail. But aggregation/refinement can only be done using models of the same type (process, product or resource). Refinements in MVP-L are level complete. That means that the refined level entirely replaces the parent level. Necessary conditions for any refinement to be level-complete are that (i) the formal parent process model interface is satisfied by the refinement, (ii) the values of each attribute of the parent process model are completely defined in terms of attributes at the refinement, and (iii) the entry and exit criteria at both levels are compatible[1].

Because the refinement of processes has the most inherent complexity of all models, it is first described completely. Thereafter the differences in product and resource refinements with respect to processes are described briefly.

2.1. Process Refinement

Process refinement is done to split a task and delegate the less complex subtasks to simpler processes. Here we use the term 'parent' to name the aggregating process, whereas the term 'child' is used to reference one of the aggregated processes. Firstly, the types and objects that are needed must be described. Therefore, <imports> and <objects> clauses have to be used. In the body only the identifiers declared and the objects declared in the process interface may appear.

<process_refinement> ::= **refinement** <imports> <objects>
<process_object_relations>
<interface_refinement> <interface_relations>
<attribute_mappings> .

Secondly, the user must describe how the aggregate object is built using objects declared in the process body. This is done by using <process_object_relations>. In this way the internal structure of aggregated elements is hidden from the other models or objects, because the objects declared in the <object> clause are not visible outside the model. In <object_relations> in a process declaration we distinguish between referencing objects and specifying alternative compositions of process behavior.

<process_object_relations>::= **object_relations** { <object_rel> ";" } .

<object_rel> ::= (<object_id> "(" { <connect> // "," }+ ")") |
("(" <structure_rel> ")") |
("(" <process_control_flow_rel> ")") .

To establish relations between objects, the model builder must specify actual parameters and bind them to formal parameters of the objects on the next lower abstraction level (using the first part of the <object_rel> rule). This is done using an object identifier declared in the <objects> clause. The following non-empty connection list contains pairs of object identifier and object values. The left hand side of each <connect> references a formal parameter of the model and the second identifier references the actual parameter, which is an <object_value>. This is an identifier declared in the <object> clause of the parent (e.g., "object") or a reference to a component of it (e.g., "object.component"). Other identifiers are not allowed to appear in a <connect> clause. The information is shared, that means that it is referenced and not copied.[1] Modifying an attribute value of a referenced object means that this change is visible for all other objects, even the old value is lost for all referencing attributes.

<connect> ::= <object_id> "=>" <object_value> .

The other way of expressing relations between objects declared in the <objects> clause is to specify the process control flow relations or product structure relations. But in contrast to product

1. Compatible means here that the set of project states matching the criteria of the process models of the aggregate level is equivalent to the set of project states matching the criteria of the process models of the refined level.

1. Usually this is known as 'pointer semantics'.

models and resource models, the first one may only appear in process model declarations. In an <object_composition> clause one can describe both relations between products and processes. Documenting the structure of products is described below. Here only the means of specifying process breakdown structure is explained.

There are three operators to specify the relations between the processes which comprise the refinement of their parent, namely "&", "|", and "exp". The operator "&" has priority over the operator "|". The operator "&" means that both operands contribute to their parent's behavior, The operator "|"explains that either the one or the other operand is enacted, and lastly the operator "exp" expresses the multiple instantiation of the objects building the operand (see below). The exponent's type must be integer, or it is specified as open value ("*"); in this case the user has to specify the value at runtime. In this way MVP-L provides for a complex specification of a process's refinement.[1]

```
<process_control_flow_rel>::=   <object_composition> .
<object_composition>      ::=   <and_composition> [ "|" <and_composition> ] .
<and_composition>         ::=   <object_composition> [ "&"
                                <object_composition> ] .
<object_factor>           ::=   ( "(" <object_id> exp <exponent> ")" ) |
                                ( "(" <object_composition> ")"
                                exp <exponent> ) |
                                ( "(" <object_composition> ")" ) | <object_id> .
<exponent>                ::=   <integer> | "*" | <object_id> .
```

The first two steps were to declare the objects a process is refined into and to specify the relations between them. Thirdly, it is important to note that a process refinement typically requires a refinement of the process interface as well. In imported models only the types of objects that are aggregated are expressed. But here the instances of the slots of the objects introduced in the <object> clause must be specified. Therefore, one can specify object composition in the same way as in the <object_relations> clause, except that the <object_id> on the left side of the <interface_refinement> has to be an object declared in a <export> or a <consume_produce> clause of the interface of the same process. In a second step the relations between the aggregated processes (children) and the aggregating process (parent) must be described. Here the same mechanisms are used as in <object_relations>. The left identifier in the connect list has to be a declared object in the parent, whereas the identifier right of "=>" has to be an object declared in the <export> or <consume_produce> clause of the child specified by the <process_object_id>.

In case of an aggregation of processes with the "exp" operator the execution semantics of an interface refinement are not prescribed by the language. The user will be asked by the runtime system to specify the distribution of the values. The same value can be given to all aggregated objects (e.g., a file name) or it can split (e.g., budget).[2]

---

1. The syntax definition now encodes the priority of both operators "|" and "&".
2. This situation is not satisfactory. It will be improved as soon as we will know more about modeling different levels of process abstractions and their connections/relations.

<interface_refinement> ::= **interface_refinement**
{ <inter_refinement> ";" } .

<inter_refinement> ::= <object_id> "=" "(" <object_composition>")" .

<interface_relations> ::= **interface_relations**
{ <interface_relation> ";" } .

<interface_relation> ::= <process_object_id>
"(" { <connect> // "," }+ ")" .

Lastly, the user must express how the information stored in attributes is shared between objects on different levels of abstraction using attribute mappings. There are two kinds of attribute mappings. In the first, the values of the children are simply added by using <sum_mapping>; this is only allowed for attribute types integer and real, or attribute models with the attribute types integer and real. In the other, the value of an attribute is determined by conditional expressions which must be disjunctive. Every time such an attribute is accessed, the expressions are evaluated. Their value must be boolean. If one expression is evaluated to **true**, the related value is the value of the mapped attribute. If no such expression exists, the value is taken from the <attr_value_map> labeled with the keyword **other**. If a default value is missing, the attribute's value is undetermined. Two lines in one <attribute_mappings> with keyword **other** are not allowed.

As for interface refinements, the semantics of attribute mappings of aggregates created with the operator "exp" are not yet defined.

<attribute_mappings> ::= **attribute_mappings** { <attr_mapping> } .

<attr_mapping> ::= <sum_mapping> | <iff_mapping> .

<sum_mapping> ::= <object_id> ":=" <sum_list> ";" .

<sum_list> ::= { <object_value> // "+" }+ .

<iff_mapping> ::= <object_id> ":" { <attr_value_map> }+ .

<attr_value_map> ::= <operand2>
"<->" (<expression> | **others**) ";" .

2.2. Product Refinement

The refinement of a product is much less complex than the refinement of a process because no delegation of behavior has to be specified. Only constructs for specifying the refinement of a model's structure are provided.

In a product's <objects> clause only instances of product models are declared. These instances are building blocks to appear in different refinement specifications. Relating objects differs somewhat from the above explanations about process model bodies. Connecting objects is exactly the same as described above. However, one is only allowed to use <product_structure_rel> clauses and one is not allowed to use <process_control_flow_rel> clauses in product refinement specifications. This is obvious because referencing instantiated process models in product models is not possible. The semantics of an object composition is the same as described above. The "&" operator explains that the product consists of both parts, "|" means that either the one or the other is part of the aggregating product. "exp" expresses that a product consists of a set of objects of the same

type, for example a program consists of several modules. The mapping of attributes works the same
in product and process models.

<structure_rel> ::= <object_composition> .

<product_refinement> ::= **refinement** <imports> <objects>
object_relations <structure_rel> ";"
<attribute_mappings> .

2.3. Resource Refinement

The specification of resource models is similar to the refinement of process models and product models. There are the same constructs as for product body specification, except that in the <object_relations> clause only connections are allowed. Similar to products in the <objects> clause, no model types other than resource models are permitted.

<resource_refinement> ::= **refinement** <imports> <objects>
object_relations <structure_rel> ";"
<attribute_mappings> .

5.5.1.3 Process Models

Project support systems have to support more than just knowledge about products. The procedures of developing and maintaining software must also be made explicit [22]. In MVP-L, process models represent knowledge of how to develop or maintain software products in a given environment. Capturing this process knowledge explicitly builds a foundation for process and product improvement.

Process models start with the keyword **process_model** followed by a unique identifier (the name of a process model) and an instantiation parameter list. The keyword **is** separates this heading from the rest of the declaration. The heading describes the important part when instantiating an object of this model. The rest of a process model declaration describes the interface components visible for use in other process model specifications, the behavior of the process, and the assignment of resources to perform the described task. Described above are interface (see 5.5.1.1 "Model Interface" on page 30) and body (see 5.5.1.2 "Model Body" on page 32) of a process model.

<process_model> ::= **process_model** <process_model_id>
<instantiation_parameters> **is**
[<comment>]
<process_interface> <process_body>
<process_resources>
end process_model <process_model_id> .

When an process object is created (or in other words, a process model is instantiated) it sometimes requires information about the context in which it is created. Instantiation parameters are used for passing information to an object during instantiation. The object specified in the <object_id> clause is called a *formal parameter*, and it is bound to the *actual parameters* when the object is instantiated. The expression describing the actual parameter value must type compat-

ible with the attribute type of the attribute model type. Only attribute types are allowed as formal parameters, whereas the values of actual parameters have to be type compatible with the corresponding formal parameters.

The <attr_model_id> does not have to appear in the <import> clause of the process model's interface specification. This identifier must specify a process attribute model; no other types of models are allowed to appear as type specification of a formal parameter. The name of the formal parameter is visible outside the object, it does not have to be made visible explicitly in the <exports> clause of the process model's interface specification.

<instantiation_parameters>::= "(" { <object_id> ":"
<attr_model_id>) // "," } ")" .

After describing interface and body of the process model one can describe what resources are assigned to the process object for performing the task. In MVP-L we distinguish between personnel and tool resources. Both constructs are described above (see 5.3.3 "Inclusion of Objects" on page 24).

<process_resources> ::= **process_resources**
[<comment>] <personnel_assignment>
[<comment>] <tool_assignment>
end process_resources .

A process model's specification is terminated by the keywords **end process_model** and the process model's identifier as specified in the beginning of the model's description.

The local scope of a process model is all the identifiers defined within that model (i.e., instantiation parameters; attributes defined in exports; consumed, produced, or modified products; objects defined in the context; identifiers of aggregated processes; and identifiers of resources).

5.5.1.4 Product Models

Product models describe the common structure of a number of software products to be developed or maintained within a project. It is obvious that the different evolution stages of a product require different representations, where each captures the information necessary for the processes operating on these product representations (e.g., requirements document, source code, or system manual).

The local scope of a product model is all the identifiers defined within that model (i.e., instantiation parameters, attributes defined in exports, and identifiers of aggregated products).

<product_model> ::= **product_model** <product_model_id>
<instantiation_parameters> **is**
[<comment>]
<product_interface> <product_body>
end product_model <product_model_id> .

5.5.1.5 Resource Models

Resource models are described in terms of <resource_interface> and <resource_body>. Their meaning is similar to the ones described under product models. The local scope of a resource

model is all the identifiers defined within that model (i.e., instantiation parameters, attributes defined in exports, and identifiers of aggregated resources).

```
<resource_model>    ::=   resource_model <resource_model_id>
                          <instantiation_parameters> is
                          <comment>
                          <resource_interface> <resource_body>
                          end resource_model <resource_model_id> .
```

There are some cases in which it is not clear whether an object of a software project could be seen as a resource or not. Assume that there is a document or manual produced within a project. A process uses this component, but not as input (i.e., not specified in <consume_produce> clause). Moreover this document helps perform the task. Eventually, one would prefer to model this document in the context of this process as a resource. This would require a type cast from a product model into a resource model. But such modifications of an object's categorization are not provided by MVP-L. Therefore, the above relation has to be modeled by using the <context> clause (see 5.3.3 "Inclusion of Objects").

5.5.2 Attribute Models

MVP-L has three kinds of user-defined attribute models (or quality models): **product_attribute_model**, **process_attribute_model**, and **resource_attribute_model**. They can be used to define observable characteristics of software processes, products, or resources, respectively. In addition, MVP-L has built-in[1] attribute models of type **global_attribute_model** which are used to refer to time, date, or the status of process execution. Each attribute model can be considered to consist of a definition of the <attribute_model> and the <attribute_manipulation>. The <attribute_type> defines the attribute as being one of the elementary data types such as integer, real, string, boolean, or an enumerated type. The <attribute_manipulation> of user-defined attributes must be provided by the user; the <attribute_manipulation> of built-in types is pre-defined. These manipulation sections define the modification of attribute values in terms of reactions to events.

```
<attribute_model>   ::=   <process_attribute_model> |
                          <product_attribute_model> |
                          <resource_attribute_model> |
                          <global_attribute_model> .
```

Global attribute models may be imported by any of the above described object models, whereas the other attribute models may only be imported by their related object model type (i.e., process attribute models are only imported into process models, and so on) (see page 24).

---

1. These built-in attribute models are not part of the language itself. Thus they are not described in this report. Because they will be heavily dependent on the underlying platform we do not expect that users should model them. These attribute models will be part of the MVP system which is not yet completely developed.

5.5.2.1 Process Attribute Models

Process attribute models describe aspects of interest which cannot be derived from a process's structure or behavior. They start with the reserved word **process_attribute_model** followed by a unique identifier. After the keyword **is**, an <attribute_model_type> clause is used to specify the type of values represented by those objects. Next the derivation of the attribute's value is described.

NOTE: The import clause in process attribute models is omitted in this version of the language. We decided not to require the import of an instantiation parameter's type. This resulted in an import clause that was always empty, which encouraged us to remove this construct from an attribute specification.

<process_attribute_model>::= **process_attribute_model** <attr_model_id>
<instantiation_parameters> **is**
[< comment>]
<attribute_model_type>
<attribute_manipulation>
end process_attribute_model
<attr_model_id> .

Only simple types or enumerations of symbols are allowed as attribute types. The standard built-in types are integer, real, boolean and string. The anonymous enumeration type is specified by listing symbols, separated by commas. If an attribute model type is used in an expression, attention must be paid to type compatibility (see 5.4.2 "Operators").

<attribute_model_type> ::= **attribute_type** [< comment>] <type> ";" .

<type> ::= <attribute_type> | "(" { <symbol> // "," }+ ")" .

After specifying attribute type and eventually importing other model types, the description of how to compute an attribute's value starts with the keyword **attribute_manipulation**. In general, we distinguish between modifications made by user request or changes forced by events released during project enactment. Because attribute manipulation is a complex part of an attribute model, it is described in an extra section (see 5.5.2.5 "Updating Attribute Values" on page 42).

<attribute_manipulation> ::= **attribute_manipulation**
[**user_triggered** { <user_manipulation> }]
[**model_triggered** { <model_manipulation> }] .

Every specification of an attribute model must terminate with the keywords **end process_attribute_model** and the attribute model's identifier.

5.5.2.2 Product Attribute Models

Product attribute models are extremely similar to process attribute models. They characterize a software component in terms of simple typed values.

NOTE: The import clause in product attribute models is omitted in this version of the language. We decided not to require the import of an instantiation parameter's type. This resulted in an

import clause that was always empty, which encouraged us to remove this construct from an attribute specification.

<product_attribute_model>::= **product_attribute_model** <attr_model_id>
<instantiation_parameters> **is**
[< comment>]
<attribute_model_type>
<attribute_manipulation>
end product_attribute_model
<attr_model_id> .

5.5.2.3 Resource Attribute Models

Resource attribute models are also extremely similar to process attribute models or product attribute models. They characterize entities of a software project or of a project environment which are involved to create or maintain a product.

NOTE: The import clause in resource attribute models is omitted in this version of the language. We decided not to require the import of an instantiation parameter's type. This resulted in an import clause that was always empty, which encouraged us to remove this construct from an attribute specification.

<resource_attribute_model>::= **resource_attribute_model** <attr_model_id>
<instantiation_parameters> **is**
[< comment>]
<attribute_model_type>
<attribute_manipulation>
end resource_attribute_model
<attr_model_id> .

5.5.2.4 Global Attribute Models

Global attribute models characterize parts or elements of the project environment which are not captured by the project description or not capturable by MVP-L. Usually they are used to provide necessary additional information during project enactment or post-mortem analysis. For example, often it would be useful to integrate the aspect of time into a process model.[1] Nevertheless, global attribute models should be used carefully in project descriptions.

<global_attribute_model> ::= **global_attribute_model** <attr_model_id> **is**
[< comment>]
<attribute_model_type>
<attribute_manipulation>
end global_attribute_model
<attr_model_id> .

---

1. This should be done with respect to MVP-L's capabilities. MVP-L is not intended to formalize time management aspects.

5.5.2.5 Updating Attribute Values

Modifications of objects during project enactment describe the progress of a software development or maintenance task. They transform a project's state into a new one. These modifications must be propagated to all objects of the project. Also a project's state can change without affecting representations of real-world elements, for example when a process is invoked by a user. The decision to enact the process should be documented. It should change the state of the process representation. In both cases the information carrier is called an event.

Modification of an attribute's value can only be triggered by an event. MVP-L has limited power for specifying events. This element of software process modeling is nearly completely hidden and is left to an implementation of a suitable MVP enaction mechanism.

There are two kinds of events related to user interaction with the MVP-L runtime system. Firstly, a general event is released, a so-called invocation. The user tells the system that a special process has to change its status. The semantic of this invocation (i.e., the reason for activating the specified process) is not known to the MVP system and therefore it is difficult to provide support for the user. These events are identified by a unique, built-in event name (e.g., **start**). Secondly, there are special events related to modifications of products. Because the user is part of the enaction mechanism for project evolution, some state transitions have to become known to the whole system for a variety of reasons (i.e., consistency checks, enabling of entry criteria, availability of objects, etc.). Therefore, starting or completing a process which consumes or produces a product has to be announced by user activity. An adequate reaction on those events can be specified in a <conditional_reaction> clause. The <default_reaction> specifies all cases not defined explicitly.

<user_manipulation> ::= **when** <event> { <conditional_reaction> | <default_reaction> }+ .

<event> ::= <invoc_id> | <system_invocation> .

<system_invocation> ::= [<process_rel> "."] <process_invoc> .

<process_rel> ::= **consume** | **produce** | **consume_produce** .

<process_invoc> ::= **start** | **complete** .

Internal events are released and managed by the runtime system of MVP-L. Syntactically they do not differ from user released events. Definition of the list of allowed event identifiers for usage in the <invoc_id> clause is left to the implementor of the enaction mechanism, e.g. as part of a library.

<model_manipulation> ::= **when** <event> { <conditional_reaction> | <default_reaction> }+ .

In the definition of reactions to a particular event, it is often useful to specify either more than one resulting state or conditions to permit only intended state transitions. Therefore, a boolean expression may express the validity of reactions. This expression is evaluated at the time that an event becomes known to the object to which an instance of the attribute model is related. In the case of more than one <conditional_reaction>, the list is evaluated from the first to the last. The first expression evaluated to **true** indicates the interpretation or execution of the related reaction state-

ment. All remaining conditionals and the default reaction are ignored. The manipulation of the attribute value as a reaction to a particular event terminates.

<conditional_reaction> ::= **and** <expression> "->" <react_stmt> .

<default_reaction> ::= **and others** "->" <react_stmt> .

The keyword **null** specifies no action. The attribute's value is left unchanged. The modification of the attribute's value is described as an assignment. The identifier in the <object_id> clause has to be the attribute model's identifier. The right hand side of the assignment statement can be either an expression or a request. In the first case, the value is computed and the result type has to be type compatible with the attribute type. In the case of a request, the user must provide the value to be assigned to the attribute. A request must appear alone on the right hand side.

<react_stmt> ::= (<assign_stmt> | **null**) ";" .

<assign_stmt> ::= <object_id> ":=" (<expression> | <mail_request>) .

5.6 Project Plans

From the model builders' point of view, a special software process model reflects particular knowledge and understanding of a special environment and its procedures applied to develop or maintain software. Therefore, it does not make sense to understand single process description components in isolation. A comprehensive analysis always requires a context. Project plans provide the needed context. This section discusses the MVP-L representation of project plans, with emphasis on the instantiation of process objects within a project plan and project plan enaction.

We have stated more than once that one should carefully distinguish between models (types) and objects. Capturing a project's data and behavior in terms of models always means documenting only the static nature of a software process. However, support of a specific project requires additional information that must be passed to instances of the models. Therefore, descriptions of projects are divided into two parts. First, a set of project plans describes the connection of environment values and object attributes. Secondly, a list of MVP-L entities describes the abstract knowledge of how to perform this project, i.e. the explicit representation of products, processes, and resources as models.

<description> ::= [<comment>]
{ <project_plan> } { <mvp_entity> } .

Software process models are instantiated and interpreted in the context of a <project plan>. A <project_plan> is described in terms of <imports>, <objects>, and <plan_objects_relations> clauses. The <imports> clause lists all models used to declare the process, product, and resource objects making up the project plan. The objects are declared in the <objects> clause.

The objects of the project plan are interconnected in the <plan_objects_relations> clause according to their formal interfaces. The <object_id> of each <plan_object_rel> must be a process identifier declared in the project plan's <objects>-part. This differs from an object relation in a process model where complete objects and parts of them may be passed to the instantiated processes. Here complete objects may be passed.

A project plan is interpreted and the processes are enacted by one or more appropriate agents. An agent may be a human or a machine.

```
<project_plan>            ::=  project_plan <project_id> is
                               <project_body>
                               end project_plan <project_id> .

<project_body>            ::=  [ <comment> ]
                               <imports> <objects> <plan_object_relations> .

<plan_object_relations>   ::=  object_relations { <plan_object_rel ";" } .

<plan_object_rel>         ::=  <object_id> "(" { <plan_connect> // "," }+ ")" .

<plan_connect>            ::=  <object_of_a_process> "=>"
                               <object_of_the_plan> .
```

The <mvp_entity> clause contains all models used in the project description; i.e., all models that are imported either by the project plan or by other models. All models imported by a project plan must appear as a MVP entity.

There are no language concepts provided for import of models from libraries or some other external store. We know too little about the reuse of software project description components, therefore this version of MVP-L lacks explicit support for model integration. Nevertheless, we are convinced that adequate reuse of such components requires an interactive tool or browser because complex project knowledge seems difficult to manage.

```
<mvp_entity>              ::=  <process_model> | <product_model> |
                               <resource_model> | <attribute_model> .
```

5.7 Key Words

The following list (i.e., Table 5) contains all reserved words in MVP-L. They may not appear as user-defined identifiers, e.g. a model's identifier.

and	attribute_manipulation	attribute_mappings
attribute_type	boolean	call
complete	consume	consume_produce
context	end	entry_exit_criteria
exp	exports	false
global_attribute_model	global_entry_criteria	global_exit_criteria
global_invariant	implementation	imports
in	integer	interface_refinement
interface_relations	is	local_entry_criteria
local_exit_criteria	local_invariant	model_triggered
not	null	object_relations
objects	or	others
personnel_assignment	process_attribute_model	process_body
process_interface	process_model	process_resources
produce	product_attribute_model	product_body
product_flow	product_interface	product_model
project_plan	real	refinement
request	resource_attribute_model	resource_body
resource_interface	resource_model	start
string	true	tool_assignment
user_triggered	when	xor

Table 5: Reserved Words in MVP-L

5.8 Summary of MVP-L Syntax

The following rules describe the syntax of sentences in MVP-L. Each syntax rule is preceded by a number. The numbering is straightforward, but sometimes the identifier of a rule is a number followed by a letter (e.g., (55a)). This means the rule has been added to the syntax definition. We did not renumber the rules but kept the same identifiers as in the previous version of this language report. This is done because the identifiers are referenced in other documents (e.g., system documentation of MVP-L support tools).

(1) <description> ::= [<comment>]
{ <project_plan> } { <mvp_entity> } .

		Project Models	
(2)	<project_plan>	::=	**project_plan** <project_id> **is** <project_body> **end project_plan** <project_id> .
(3)	<project_body>	::=	[<comment>] <imports> <objects> <plan_object_relations> .
		Model Types	
(4)	<mvp_entity>	::=	<process_model> | <product_model> | <resource_model> | <attribute_model> .
		Process Models	
(5)	<process_model>	::=	**process_model** <process_model_id> <instantiation_parameters> **is** [<comment>] <process_interface> <process_body> <process_resources> **end process_model** <process_model_id> .
(6)	<process_interface>	::=	**process_interface** [<comment>] <import_export> <product_flow> <context> <criteria> **end process_interface** .
(7)	<process_body>	::=	**process_body** ([<comment>] <process_refinement> | <process_implementation> [<comment>]) **end process_body** .
(8)	<process_resources>	::=	**process_resources** [<comment>] <personnel_assignment> [<comment>] <tool_assignment> **end process_resources** .
(9)	<process_refinement>	::=	**refinement** <imports> <objects> <process_object_relations> <interface_refinement> <interface_relations> <attribute_mappings> .
(10)	<process_implementation>	::=	**implementation** .
(11)	<personnel_assignment>	::=	**personnel_assignment** [<imports> <objects>] .
(12)	<tool_assignment>	::=	**tool_assignment** [<imports> <objects>] .
		Product Models	
(13)	<product_model>	::=	**product_model** <product_model_id> <instantiation_parameters> **is** [<comment>] <product_interface> <product_body> **end product_model** <product_model_id> .

(14) <product_interface> ::= **product_interface**
[<comment>] <import_export>
end product_interface .

(15) <product_body> ::= **product_body**
([<comment>] <product_refinement> |
<product_implementation> [<comment>])
end product_body .

(16) <product_refinement> ::= **refinement** <imports> <objects> **object_relations**
<structure_rel> ";" <attribute_mappings> .

(17) <product_implementation> ::= **implementation** .

Resource Models

---

(18) <resource_model> ::= **resource_model** <resource_model_id>
<instantiation_parameters> **is** [<comment>]
<resource_interface> <resource_body>
end resource_model <resource_model_id> .

(19) <resource_interface> ::= **resource_interface**
[<comment>] <import_export>
end resource_interface .

(20) <resource_body> ::= **resource_body**
([<comment>] <resource_refinement> |
<resource_implementation> [<comment>])
end resource_body .

(21) <resource_refinement> ::= **refinement** <imports> <objects> **object_relations**
<structure_rel> ";" <attribute_mappings> .

(22) <resource_implementation> ::= **implementation** .

Attribute Models

---

(23) <attribute_model> ::= <process_attribute_model> |
<product_attribute_model> |
<resource_attribute_model> | <global_attribute_model> .

(24) <process_attribute_model> ::= **process_attribute_model** <attr_model_id>
<instantiation_parameters> **is** [<comment>]
<attribute_model_type> <attribute_manipulation>
end process_attribute_model <attr_model_id> .

(25) <product_attribute_model> ::= **product_attribute_model** <attr_model_id>
<instantiation_parameters> **is** [<comment>]
<attribute_model_type> <attribute_manipulation>
end product_attribute_model <attr_model_id> .

(26) <resource_attribute_model> ::= **resource_attribute_model** <attr_model_id>
<instantiation_parameters> **is** [<comment>]
<attribute_model_type> <attribute_manipulation>
end resource_attribute_model <attr_model_id> .

(27) <global_attribute_model> ::= **global_attribute_model** <attr_model_id> **is**
[<comment>] <attribute_model_type>
<attribute_manipulation>
end global_attribute_model <attr_model_id> .

(28) <attribute_model_type> ::= **attribute_type** [<comment>] <type> ";" .

(29) <attribute_manipulation> ::= **attribute_manipulation**
[**user_triggered** { <user_manipulation> }]
[**model_triggered** { <model_manipulation> }] .

(30) <type> ::= <attribute_type> | "(" { <symbol> // "," }+ ")" .

(31) <attribute_type> ::= **integer** | **real** | **string** | **boolean** .

Attribute Manipulations

---

(32) <user_manipulation> ::= **when** <event> { <conditional_reaction> |
<default_reaction> }+ .

(33) <model_manipulation> ::= **when** <event> { <conditional_reaction> |
<default_reaction> }+ .

(34) <event> ::= <invoc_id> | <system_invocation> .

(35) <system_invocation> ::= [<process_rel> "."] <process_invoc> .

(36) <assign_stmt> ::= <object_id> ":=" (<expression> | <mail_request>) .

(37) <process_rel> ::= **consume** | **produce** | **consume_produce** .

(38) <process_invoc> ::= **start** | **complete** .

(39) <conditional_reaction> ::= **and** <expression> "->" <react_stmt> .

(40) <default_reaction> ::= **and others** "->" <react_stmt> .

(41) <react_stmt> ::= (<assign_stmt> | **null**) ";" .

Imports/Exports

---

(42) <import_export> ::= <imports> <exports> .

(43) <imports> ::= **imports** { <library_part> ";" } .

(44) <exports> ::= **exports** { <attribute> ";" } .

(45) <library_part> ::= <model_type> { <model_id> // "," }+ .

(46) <attribute> ::= <object_id> ":" (<attribute_type> | <attr_model_id>)
[":=" <attribute_value>] .

(47) <model_type> ::= **process_model** | **product_model** | **resource_model** |
global_attribute_model | **process_attribute_model** |
product_attribute_model
| **resource_attribute_model** .

(48) <attribute_value> ::= <expression> .

		Consume/Produce	
(49)	<product_flow>	::=	**product_flow** [<comment>] <consume> [<comment>] <produce> [<comment>] <consume_produce> .
(50)	<consume>	::=	**consume** { <object_id> ":" <product_model_id> [< selection>] ";" } .
(51)	<produce>	::=	**produce** { <object_id> ":" <product_model_id> [< selection>] ";" } .
(52)	<consume_produce>	::=	**consume_produce** { <object_id> ":" <product_model_id> [< selection>] ";" } .
(52a)	<selection>	::=	"(" { <ident> // "," }+ ")" .
		Context	
(53)	<context>	::=	**context** { <object_decl> } .
		Criteria	
(54)	<criteria>	::=	**entry_exit_criteria** [<comment>] <entry_criteria> [<comment>] <invariant> [<comment>] <exit_criteria> .
(55)	<entry_criteria>	::=	**local_entry_criteria** [<criteria_expression>] **global_entry_criteria** [<criteria_expression>] .
(55a)	<invariant>	::=	**local_invariant** [<criteria_expression>] **global_invariant** [<criteria_expression>].
(56)	<exit_criteria>	::=	**local_exit_criteria** [<criteria_expression>] **global_exit_criteria** [<criteria_expression>] .
(57)	<criteria_expression>	::=	<expression> ";" .
		Objects	
(58)	<objects>	::=	**objects** { <object_decl> } .
(59)	<object_decl>	::=	<object_id_list> ":" <type_decl> ";" .
(60)	<type_decl>	::=	<model_id> ["(" <parameter_list> ")"] .

Object Relations

(61) <process_object_relations> ::= **object_relations** { <object_rel> ";" } .

(61a) <plan_object_relations> ::= **object_relations** { <plan_object_rel> ";" } .

(62) <object_rel> ::= (<object_id> "(" { <connect> // "," }+ ")") |
("(" <structure_rel> ")") |
("(" <process_control_flow_rel> ")") .

(62a) <plan_object_rel> ::= <object_id> "(" { <plan_connect> // "," }+ ")" .

(63) <structure_rel> ::= <object_composition> .

(64) <process_control_flow_rel> ::= <object_composition> .

(65) <object_composition> ::= <object_composition>
["|" <and_composition>] .

(66) <and_composition> ::= <and_composition>
["&" <object_factor>] .

(66a) <object_factor> ::= ("(" <object_id> **exp** <exponent>")")
| ("(" <object_composition> ")" **exp** <exponent>)
| ("(" <object_composition> ")") | <object_id> .

(67) <exponent> ::= <integer> | "*" | <object_id> .

Interface Refinements

(68) <interface_refinement> ::= **interface_refinement** { <inter_refinement> ";" } .

(69) <inter_refinement> ::= <object_id> "=" "(" <object_composition>")" .

Interface relations

(70) <interface_relations> ::= **interface_relations** { <interface_relation> ";" } .

(71) <interface_relation> ::= <process_object_id> "(" { <connect> // "," }+ ")" .

(72) <connect> ::= <object_id> "=>" <object_value> .

(72a) <plan_connect> ::= <object_of_a_process> "=>" <object_of_the_plan> .

Attribute Mappings

(73) <attribute_mappings> ::= **attribute_mappings** { <attr_mapping> } .

(74) <attr_mapping> ::= <sum_mapping> | <iff_mapping> .

(75) <sum_mapping> ::= <object_id> ":=" <sum_list> ";" .

(76) <sum_list> ::= { <object_value> // "+" }+ .

(77) <iff_mapping> ::= <object_id> ":" { <attr_value_map> }+ .

(78) <attr_value_map> ::= <operand2> "<->" (<expression> | **others**) ";" .

(79)	<symbol>	::=	"'" <ident> "'" .
(80)	<object_id_list>	::=	{ <object_id> // "," }+ .
(81)	<object_id>	::=	<ident> .
(82)	<process_object_id>	::=	<ident> .
(83)	<model_id>	::=	<ident> .
(84)	<object_model_id>	::=	<ident> .
(85)	<project_id>	::=	<ident> .
(86)	<process_model_id>	::=	<ident> .
(87)	<product_model_id>	::=	<ident> .
(88)	<resource_model_id>	::=	<ident> .
(89)	<attr_model_id>	::=	<ident> .
(90)	<event_id>	::=	<ident> .
(91)	<invoc_id>	::=	<ident> .
(91a)	<object_of_a_process>	::=	<ident> .
(91b)	<object_of_the_plan>	::=	<ident> .
(92)	<instantiation_parameters>	::=	"(" { <object_id> ":" <attr_model_id>) // "," } ")" .
(93)	<parameter_list>	::=	{ <attribute_value> // "," }+ .
(94)	<comment>	::=	"{" { <character> } "}" | { "--" { <character> } }+ .
(95)	<integer>	::=	["-"] { digit }+ .
(96)	<real>	::=	["-"] { digit }+ "." { digit }+ .
(97)	<string>	::=	'"' { character } '"' .
(98)	<boolean>	::=	**true** | **false** .
(99)	<digit>	::=	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
(100)	<character>	::=	<digit> | <alpha_char> | <ident_char> | <special_char> .
(101)	<alpha_char>	::=	"A" | ... | "Z" | "a" | ... | "z" .
(102)	<ident_char>	::=	"_" | "-" .
(103)	<special_char>	::=	"!" | "@" | "#" | "$" | "%" | "^" | "&" | "*" | "(" | ")" | "+" | "=" | "{" | "}" | "[" | "]" | ":" | ";" | "'" | "~" | "<" | ">" | "." | "," | "?" | "/ " | "|" | "\" | "`" | " " .
(104)	<ident>	::=	<alpha_char> { <alpha_char> | <ident_char> | <digit> } .
(105)	<expression>	::=	<arith_exp> <compare_op> <arith_exp> | <arith_exp> .
(106)	<compare_op>	::=	"=" | "!=" | "<" | "<=" | ">" | ">=" .

(107) <arith_exp> ::= <arith_exp> "+" <arith_term> |
<arith_exp> "-" <arith_term> |
<arith_exp> **or** <arith_term> |
<arith_exp> **xor** <arith_term> |
"-" <arith_term> |
<arith_term> .

(108) <arith_term> ::= <arith_term> "*" <arith_factor> |
<arith_term> "/" <arith_factor> |
<arith_term> **and** <arith_factor> |
<arith_factor> .

(109) <arith_factor> ::= <operand> | "(" <expression> ")" |
not <arith_factor> .

(110) <operand> ::= <object_value> | <symbol> | <integer> | <real> | <string>
| <boolean> | <tool_invocation> .

(110a) <operand2> ::= <symbol> | <integer> | <real> | <string> | <boolean> .

(111) <object_value> ::= <object_id> ["." <object_id>] .

(112) <tool_invocation> ::= **call** "(" <tool_identifier>
["(" { <expression> // "," }+ ")"] ")"
[**in** (<integer_interval> | <real_interval> |
<string_enumeration> | <bool_enumeration> |
<symbol_enumeration>)].

(113) <tool_identifier> ::= <resource_model_id> .

(114) <integer_interval> ::= "[" <integer> ".." <integer> "]" .

(115) <real_interval> ::= "[" <real> ".." <real> "]" .

(116) <string_enumeration> ::= "[" { <string> // "," }+ "]" .

(117) <bool_enumeration> ::= "[" { <boolean> // "," }+ "]" .

(118) <symbol_enumeration> ::= "[" { <symbol> // "," }+ "]" .

(119) <mail_request> ::= **request** "(" <message_string> ")" .

(120) <message_string> ::= <string> .

6 Consistency Rules

In this chapter we give a summary of consistency rules which must be satisfied by all models and project plans which are to be interpreted. They were introduced implicitly in the previous section. Now we want to give a detailed description. The rules are implemented and the functions make up the component "ConsCheck" which is part of the MVP-L modeling tool [31]. This tool uses the rules' numbers for messages about consistency faults.

For each MVP-L construct we list the corresponding rules which are explained in more detail. Note that these rules describe *static* properties for internal consistency of project plans and supporting models.

We distinguish between local and global requirements. These requirements are part of the rules which affect relationships between properties of the same model or project plan (local requirements) and relationships between the models and project plans (global requirements). The first is called *intra-model consistency* and the latter is called *inter-model consistency.* The reason to distinguish between both sets of consistency rules is that it is useful for models to be reused to check the intra-model consistency before they are related to other elements in the context of a project plan.

Consistency of Identifiers

(CR1) Valid identifiers

Requirements for identifiers result mainly from the syntax rules appearing in this language report (Rule 104):

- The first character of an identifier is a letter.[1]
- Subsequent characters of an identifier may be letters, digits, "-", and "_".
- Reserved words may not be used as identifiers
  (see Table 5, "Reserved Words in MVP-L," on page 45).
- For implementation reasons the length of identifiers is restricted to the range [1 .. 128]. This limitation does not appear in the grammar.

	Example
OK	**product_model** Low_level_design_document
error	**product_model** 1_level_design_document%
reason	*first character not a letter, last character special character*

1. We use lower-case letters for object identifiers and upper-case letters for identifiers of models. This is not required by the grammar. However, we suggest this style.

(CR2) Validity of events
Six events are predefined in MVP-L: **consume.start**, **consume.complete**, **produce.start**, **produce.complete**, **consume_produce.start**, and **consume_produce.complete**. These events may only appear in user-triggered attribute manipulations of product attribute models. In addition, the user may define events. Reserved words of MVP-L and identifiers of the attribute's local scope may not be used as event names. It should be recognized that the process engine only supports a fixed set of events [32].

(CR3) Attribute model types
Valid types of an attribute model are:

- Standard type **integer**, **real**, **string**, or **boolean**.
- Enumeration of symbols.
  Symbols are identifiers (see (CR1)) enclosed in "'". A reserved word enclosed in "'" is also a valid symbol. A symbol may appear only once in a symbol list.

Consistency of Object Declarations

(CR4) Requirements with respect to the local scope
The local scope of a process, product, or resource model consists of:

- The model identifier.
- The model identifiers of imported models.
- The identifiers of objects declared in the model (i.e., identifiers of exports, product_flow, context, object names in refinement, and object names of process_resources).

The use of identifiers in a model must be unambiguous, each used identifier must be defined exactly once.

Example

```
product_model High_level_design_document (status_0: Product_status) is
    product_interface
        imports
            product_attribute_model Product_status;
        exports
            status: Product_status := status_0;
    end product_interface
    product_body
        implementation
    end product_body
end product_model High_level_design_document
```

The local scope is: High_level_design_document, status_0, Product_status, status.

error **exports**
status: Product_status := status_0;
status: Other_status;
reason *Identifier* status *declared more than once.*

The local scope of an attribute model (process, product, resource, or global attribute model) consists of:

- The model identifier.
- Events used in the attribute model.

Event names may occur only once.

(CR5) Requirements with respect to global scope
Objects in the global scope are identifiers of models and project plans only. Each identifier may occur only once.

(CR6) Use of objects

a) Use of objects in arithmetic expressions:
Arithmetic expressions and objects appear in the different models only at distinct places. Within expressions identifiers may appear as operands. Table 6 lists the rules which specify where used objects have to be declared. The left column lists the location of the expression and the right column describes where the operands appearing in that expression have to be declared. Identifiers of tools have to be valid identifiers of resource models. They need not be imported. Parameters of an actual parameter list of a tool invocation are expressions too. The same rules apply.

Arithmetic expressions and objects (Use)	Location of operand's declaration
initial values for attributes (<exports>)	• instantiation parameters (<instantiation_parameters>)
(<local_entry_criteria>, <local_exit_criteria>, <local_invariant>)	• locally defined attributes (<exports>) • attributes of local interface parameters (<product_flow>)
(<global_entry_criteria>, <global_exit_criteria>, <global_invariant>)	• attributes of the global interface parameters (<context>) (at least one such attribute must be used in a global criterion!) • locally defined attributes (<exports>) • attributes of local interface parameters (<product_flow>)
parameter in object declaration (only allowed in <objects> of project plans and process models!)	• instantiation parameter (<instantiation_parameter>)

Table 6: Declaration of Operands and Their Use in Arithmetic Expressions

Arithmetic expressions and objects (Use)	Location of operand's declaration
expressions of attribute mappings (<object_value> in <sum_list> or <expression> in <attr_value_map>	• attributes of objects from <objects> or <product_flow> • local defined attributes (<exports>)
resulting object (<object_id>) to be computed in attribute mappings	• locally defined attributes (<exports>)
expressions in <conditional_reaction> and <assign_stmt> of attribute models	• identifier of the attribute model (<attr_model_id>)
object (<object_id>) to be computed in attribute manipulations of attribute models	• identifier of the attribute model (<attr_model_id>)

Table 6: Declaration of Operands and Their Use in Arithmetic Expressions

b) Use of objects in refinements
Table 7 shows where objects used in refinements of models must be declared.

Objects	Location of Declaration
objects in <object_relations>	<objects>
interface objects in <interface_refinement>	<produce>
composition object in <interface_refinement>	<objects>
process object in <interface_relations>	<objects>
connect object in <interface_relations>	<product_flow>
exponent in <object_composition>	<instantiation_parameters>

Table 7: Objects in Refinements

(CR7) Import of models

a) Validity of model type and attribute type
The validity of an object's model type is determined by Table 2, "Import Relations Allowed between Model Types," on page 24. For the following list of objects additional requirements with respect to a model type must be considered (for example: each object specified in a product flow clause has to be a model describing a product, that means it has to be an instance of a **product_model**):

- Formal parameters appearing as instantiation parameters after the model identifier must have an attribute model type corresponding to the model (i.e., the type of a process's formal interface parameter is only allowed to be process attribute model).

- Attributes of a process, product, or resource can have simple types (**integer**, **real**, **string**, or **boolean**), or are allowed to be instances of global attribute models or an attribute model which corresponds to the type of the model containing the <export> clause (e.g., **process_attribute_model** in the case of an process attribute).
- Objects declared in the <product_flow> clause of a process are only allowed to be of type **product_model**.
- Objects declared in the <context> clause of a process are allowed to be **process_model**, **product_model**, or **resource_model**.
- Objects declared in the <objects> clause of a project plan are allowed to be of type **process_model**, **product_model**, or **resource_model**.
- Objects declared in the <objects> clause of a process model are allowed to be of type **process_model** or **product_model**.
- Objects declared in the <objects> clause of a product model are allowed to be of type **product_model**.
- Objects declared in the <objects> clause of a resource model are allowed to be of type **resource_model**.
- Objects declared in the <process_resources> clause of a process model are allowed to be of type **resource_model**.
- Objects appearing as the left-most designator in <object_relations> clause of a project plan or in a <interface_relations> clause of process models must be of type **process_model**.
- Objects appearing in the <object_composition> clause of an <interface_refinement> must be of type **product_model**.

b) Declaration of imported models
Keywords appearing in an import clause have to be identifiers of other models appearing in the project description[1]. The model type of the import clause and the type of the model referenced by the model_id have to be the same.

	Example
OK	**import** **product_model** Low_level_design_document;
error	**import** **process_model** Low_level_design_document;
reason	Low_level_design_document *is a product model*

1. MVP-L does not have any language construct to aggregate models and project plans (i.e., modules). We assume that models to be checked for consistency are grouped in a project description (i.e., which physically resides in a file).

c) No cyclic imports
Cyclic imports are not allowed in MVP-L, neither directly (a model imports itself) nor indirectly (a model is imported by the models it imports, recursively).

Validity of expressions, assignments, and parameters

(CR8) Validity of expressions and assignments

a) Expressions
The validity of an expression is checked using Table 4, "Signatures of MVP-L Operators," on page 28. To be compatible, values of symbol types must come from the same attribute model. This means the attributes have to be instances of the same attribute model. The return value type of tool calls is determined by the clause following the call or defaults to **real**.

Example

```
product_attribute_model PA1 () is
    attribute_type
        ('value_1')

product_attribute_model PA2 () is
    attribute_type
        ('value_1', 'value_2')

exports
    pa1 : PA1;
    pa2 : PA2;
```

OK (pa1 = 'value_1') and (pa2 = 'value_1')
error pa1 = pa2
reason *pa1 and pa2 are not instances of the same attribute model type.*

b) Assignments
Assignments of values to objects are only allowed if the left-hand and right-hand side have the same type; one exception here is the assignment of an integer value to objects of type real. If symbols are assigned, the left-hand and right-hand side have to be instances of the same attribute model. The return type value of a request is always assumed to be the attribute type of the attribute model which contains the request statement.

c) Special arithmetic expressions
The result type of arithmetic expressions depends on the context of the expression. The following restrictions apply:

Expression	Type
expression in <criteria>	**boolean**

Table 8: Types of special arithmetic expressions

Expression	Type
objects in <sum_mapping>	**integer** or **real**
conditions in <iff_mapping>	**boolean**
conditions in <conditional_reaction>	**boolean**

Table 8: Types of special arithmetic expressions

d) Tool invocations
Tool invocations may only appear within attribute models. The tool identifier must be a valid identifier of a resource model which must not be imported. The parameter list must be correct with respect to Rule (CR9). The return value is **integer**, **real**, **string**, or **boolean**.

(CR9) Validity of parameter list
Actual parameters of the parameter list must be type compatible (Table 4, "Signatures of MVP-L Operators," on page 28) with the corresponding formal parameters. In addition, the number of actual and formal parameters must be equal. Parameter lists are only allowed in bodies of project plans and processes. If a model has instantiation parameters, actual parameters must be provided when instantiating an object of that type.

Validity of selections

(CR10) Selections may appear as a suffix of a product declaration within the product flow specification of a process model. A selection may only use attribute identifiers declared in the export part of the related product model.

Example

```
product_model A () is
    exports
        first_attribute, second_attribute : Any_Attribute;
```

OK

```
process_model good_example () is

    product_flow
        consumes
            a : A (first_attribute);
```

error

```
process_model bad_example () is

    product_flow
        consumes
            a : A (first_attribute, third_attribute);
```

reason The name of the attribute *third_attribute* is not defined in the product model, hence it does not exist and may not be referenced in the selection.

Validity of refinements

(CR11) Validity of <object_relations>

a) <object_relations> within project plans may only contain connections.

b) For describing object relations of processes, products, and resource models, exactly one <object_composition> clause is allowed.

(CR12) Validity of <object_relations> within project plans
Project plans must pass products or resources to process objects without refining them, even if they are complex. This is now ensured by a new syntax rule (see (72a)).

a) Local requirements (intra-model consistency)

- The process object of each <object_relations>-statement (i.e., the left-most identifier) must be instantiated in the <objects> clause of the project plan.
- The type of this object has to be **process_model**.
- All objects used as the right part of a <plan_connect> (i.e., <object_of_the_plan>) must be declared in the <objects> clause of the project plan (see also (CR6a)).

Example

```
project_plan a_project is
    imports
        product_model Any_product_model;
        process_model Any_process_model;
    objects
        a_process : Any_process_model;
        a_product : Any_product_model;
```

OK

```
    object_relations
        a_process (input => a_product);
```

error

```
    object_relations
        a_product (component => not_declared_product);
```

reason a_product is not a process and the connected object is not declared as an object of the project plan.

- For each process declared in the <objects> clause of the project plan there must be an <object_relations>-statement.
- All products and resources declared in <objects> must be connected.

b) Global requirements (inter-model consistency)
The following constraints consider information defined in the models used in a project plan:

- The model type of each process object must exist.

- All objects declared in the model type of an instantiated process as <product_flow> or <process_resources> must be connected to an object of the project plan exactly once. Other objects of a process model type may not appear in a <object_relations>-statement.
- Each product object of the project plan may be produced exactly once, that means it may be only connected to one produced object of a process object.
- Objects connected in <object_relations> must be instances of the same model type.

(CR13) The <object_composition>-statement

- The combination of objects to form more complex structures of a 'is-part-of'-hierarchy is done using the operators "&", "|", and "*". In the following we distinguish between *known* and *unknown* exponents. Known exponents are positive integers. Unknown exponents are identifiers or the "*"; their value is not known for static tests.
- A composition statement is allowed only to specify those compositions in which an object appears at most once. A composition is described using only **&** and **exp** (e.g., the composition statement (a & (b | c)) describes two compositions, namely (a & b) and (a & c)).

	Example
OK	(a & b & c)
	((a & b) | (a & c))
error	(a & a & b)
	(a & (a | b)) = ((a & a) | (a & b))
reason	*Object a appears two times in the same composition.*

- Validity of <object_composition>-statements within <interface_refinement>
  A refinement of a process interface uses object compositions to specify the aggregation of produced product objects. If the process is performed, the product is refined/aggregated as expressed in the composition statement. The resulting structure must be compatible with the structure specified in the product model type.

Example

```
product_model Prod () is
    objects
        a : any_product;
        b : another_product;
        c : yet_another_product;
    object_relations
        ((a & b) | (b & c))
end product_model Prod

process_model Proc () is
```

	produce i1, i2: Prod; **objects** x : any_product; y : another_product; z : yet_another_product; **interface_refinement**
OK	i1 = (x & y)
error	i2 = (x & y & z)
reason	*The aggregation/refinement of i2 is not specified in the product model.*

(CR14) Validity of <object_relations>-statement in process, product, and resource models

- Each object used in a composition statement of a model must be declared in the <objects> clause of the model (see also (CR6a)).
- The <object_composition>-statement may contain only objects with the same type as the model, that means only process objects may appear in process model compositions, product objects in product model refinements, and resource objects in resource model refinements. Each object specified in <objects> of the refined model has to be used in an <object_composition>-statement.
- Unknown exponents of type <object_id> must appear as instantiation parameters. The type of those parameter must be integer.

(CR15) Validity of <interface_refinement>-statements in process models (Extension of (CR14) for process models)

a) Local requirements (intra-model consistency)

- Interface objects to be refined must appear in <produce> (see also (CR6a)).
- Each interface object is refined at most once.
- Objects appearing in the composition must be declared in the process's <objects-clause and to be of model type **product_model**.
- Used exponents must appear in the instantiation parameters.
- The object composition must be consistent with respect to (CR13).
- All products declared in the <objects> clause must be used in the refinement.

b) Global requirements (inter-model consistency)

- An exponent's type must always be **integer** (see also (CR13)).
- The composition of an interface object must be compatible with the refinement specified in the product's refinement (see also (CR13)).

(CR16) Validity of <interface_relations>-statements within process models
The <connect>-statements within process models may refer to an object or to a part of it (in the case of aggregated products or resources). The following restrictions apply:

Balanced product flow

- Each consumed product of the refined process or, if the product is an aggregated one, one of its components must be consumed at least by one subprocess. The product or its components may neither be produced nor modified (conume_produce'd) by one of the subprocesses.
- Each modified product of the refined process or, if the product is an aggregated one, one of its components must be modified at least by one subprocess. The product or its components may not be produced, but consumed by the subprocesses.
- Each produced product of the refined process or, if the product is an aggregated one, one of its components, is produced exactly by one subprocess.

Validity of product flow

- Each consumed or produced product has to be passed either as a whole or in parts to the subprocesses; no mixing of aggregates and components of a product is allowed within a connect or in different connects.

Example
interface_relations
error a_process (a_process_i1 => product, a_process_i2 => product.part)
reason *Both a complex product and one of its parts are passed to a subprocess (a_process). The object "product" has to be declared in the process's <consume> or <consume_produce>.*

- The set of all product parts passed to a particular subprocess has to be a subset of a valid refinement of the aggregating product.

Example
Refinement of product model A is
object_composition
(b | c)

process_model P () is
refinement
objects
a : A;
s : SubProcess;
interface_relations
error s (a.b => param1 , a.c => param2);
reason *The <interface relations>-statement requires a "&"-composition of the passed product parts, the product specification defines a "|"-composition.*

- The set of product parts produced by subprocesses has to be a valid refinement of the aggregating product. In addition, in the case of an interface refinement, there must be exactly one process to produce one part of the product.

Example
Refinement of product model A is
object_composition
(b & c) -- b and c are both instances of Type_of_b_and_c

Product flow of product model SubProcess is
product_flow
consume
produce
param1 : Type_of_b_and_c;
consume_produce

process_model P () is
refinement
objects
a : A;
s1, s2 : SubProcess;

OK
interface_relations
s1 (param1 => a.b);
s2 (param1 => a.c);

error
interface_relations
s1 (param1 => a.b);
s2 (param1 => a.b);

reason *Product a's part b is produced twice by the subprocesses, part c is not produced.*

- All interface refinements have to reflect the same product structure.

Example
Refinement of product model A is
object_composition
(b & c) | (b & d)

process_model P () is
refinement
objects
a : A;
s1, s2 : SubProcess;

interface_relations

error

```
s1 ( param1 => a.b, param2 => a.c );
s2 ( param1 => a.b, param2 => a.d );
```

reason *Passing a's parts to s1 and s2 requires different refinements of a.*

The following additional requirements for interface relations within process models must also be fulfilled:

a) Local requirements

- The subprocess of each <interface_relation>-statement has to be declared in the refined process's <objects>-statement. The model type has to be **process_model**.
- For each process object appearing in the refined process's <objects>-statement there must exist a <interface_relation>-statement.
- All products appearing in a <connect>-statement must be declared in the process's <product_flow>-statement.
- All products appearing in the <product_flow>-statement must be used by at least one <interface_relation>.

b) Global requirements

- The process models of the subprocesses must exist (CR7).
- The parameters of the subprocesses must exist within the subprocesses' <product_flow>-statements.
- Each interface product of a subproduct must be connected exactly once.
- The product flow has to be balanced and valid according to the above defined rules.
- Connected products (actual and formal parameters) have to be type compatible.

(CR17) Refinement of attributes in attribute mappings
The value of each attribute appearing in a refined product's, process's, or resource's <export>-statement must be defined by using the attributes of the subproducts, subprocesses, or subresources.

(CR18) Compatibility of criteria
The criteria of the refined process and those of its subprocesses have to be compatible. In particular, this means:

a) All subprocesses' entry criteria are logically or-ed. This boolean condition describes a set of states. Between this set and the set of states determined by the entry criterion of the refined process, the subset-relation must hold. The subprocesses may describe a larger set of states than the refined process.

b) All subprocesses' invariants are logically and-ed. This boolean condition describes a set of states. This set and the set of states determined by the invariant of the refined process must be equal.

c) All subprocesses' exit criteria are logically and-ed. This boolean condition describes a set of states. The subset-relation between the set of states determined by the exit criterion of the refined process and this set must hold. The subprocesses may describe a smaller set of states than the refined process.

Validity of MVP-L language constructs

(CR19) Validity of <iff_mapping>-statements
A <iff_mapping>-statement may contain at most one default branch (**others**). There has to be at least one <attr_value_map>-statement in the mappings.

(CR20) Validity of <user_manipulation>- and <model_manipulation>statements
A <user_manipulation>- or <model_manipulation>-statement may contain at most, but not necessarily one default branch (**others**).

(CR21) Process, product, or resource attribute models may contain no instantiation parameters. They are instantiated by assignment in the process, product, or resource model.

7 Conclusions and Future Directions

MVP-L has been designed to support the creation of descriptive process models, the creation of prescriptive project plans, the analysis of project plans, the packaging of process models for reuse, the enactment of project plans, and the documentation of project plan enactment histories. The language features emphasize modeling in-the-large, natural modeling, and a formal execution model.

We have validated the language MVP-L in several artificial and real process scenarios, including the modification process designed for the 6th International Software Process Workshop [19], the maintenance process at NASA's Software Engineering Laboratory [28], and the improvement of a reuse process employed within TRW's STARS division [18]. Experience from these applications have been used to refine the language further.

We work towards a MVP-S system supporting the construction, analysis and execution guidance of project plans (see Figure 5).

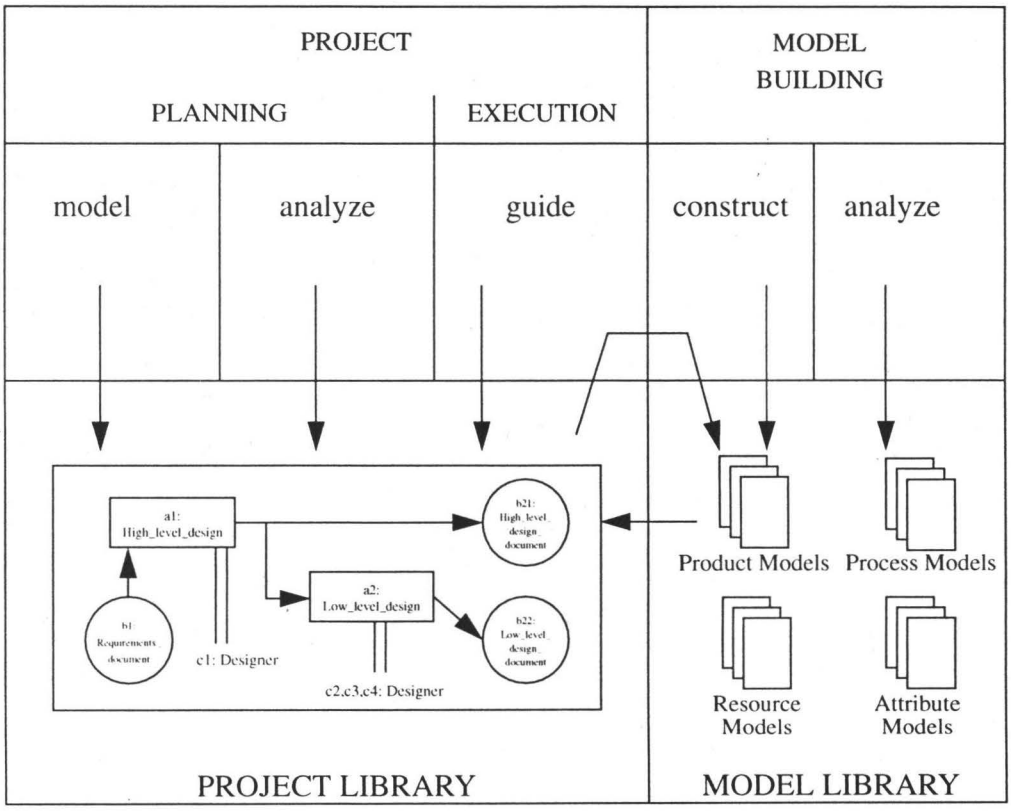

Figure 5: Architecture of MVP-S

The MVP-S system is intended to support not only the work of individual programmers, but especially the cooperation between individuals. According to the classification scheme published by Perry and Kaiser, our environment is at the 'family' level [23]. Current work in the MVP project focuses on the development of methods for descriptive modeling (especially the creation of models from multiple views of a process), the design and implementation of analysis algorithms, the

definition and implementation of a system for executing MVP-L project plans, the design of algorithms for resolving conflicts occurring during execution, and the establishment of a model hierarchy within the model library based on multiple inheritance.

Currently, a process engine exists for interpreting MVP-L process descriptions [32]. It transforms project states and interacts with the user via a graphical user interface. This tool supports the project organization by tracking the project state, storing relevant information, and informing the user about changed project states. Guidance is provided by taking measurement data and evaluating entry and exit criteria.

Among other automated support of the software process modeling and enaction task we will develop a multi-view modeling tool with consistency checking , an analysis tool for investigating existing models statically and dynamically, and automatic help to establish connections between attributes of process entities and measurement tools. At the moment a collection of tools is available for editing MVP-L files (i.e., syn-dir-ed, a syntax-directed editor generated by the Cornell Synthesizer Generator), inserting them into an internal representation (i.e., MVP-L-i), checking the models for consistency (i.e., ConsCheck) [31], modifying information stored in the internal representation about product flow (i.e., ProdFlowEd), and storing them in a relational database (i.e., MVP-L-i_2_INGRES). They all are integrated by a graphical user interface (i.e., MoST - Modeling Support Tool) [6].

8 What has changed since earlier versions of MVP-L

We are aware that MVP-L is a prototype language that lays foundations for investigations in the area of software process modeling. Therefore, improvements in the language definition have to be made as soon as results from modeling approaches provide feedback from the language users. First attempts in building larger software process models were the solution for the 6th ISPW [19] and describing a reuse scenario at TRW [18]. These activities and experience gathered iteratively from other observations necessitated some changes to the language as described in earlier publications. Here a list is given of what has changed with respect to the version of MVP-L described in [26] and why these changes were made. These brief explanations should be a help for long-time users of MVP-L.

1. Consume and Produce for Products

The specification of a process's behaviour concerning consuming and producing products now starts with the keyword **product_flow**. Also the order of terminal symbols in the declaration has changed: First, the keywords **consume** or **produce** have to appear, and then the object(s) followed by their type. Additionally, the clause <consume_produce> was introduced to allow for documenting modification or version change of an object.

Earlier versions of MVP-L:

```
consume_produce
    o1 : consume any_document;
    o2 : consume another_document;
    o3 : produce output;
    o1 : produce any_document
```

Now:

```
product_flow
    consume o2 : another_document;
    produce o3 : output;
    consume_produce o1 : any_document;
```

2. Entry and Exit Criteria

The keywords **local_entry_criteria**, **global_entry_criteria**, **local_exit_criteria**, and **global_exit_criteria** must appear, even if no related criteria is specified.

3. Attribute Manipulations

The attribute manipulations in attribute model declarations have changed substantially. Attribute values now change purely by reacting to events, or when the user sets a new value manually. Assignments describing events (e.g., *entry := true* for specifying that the precondition of the process to which the attribute is attached to has become true) are no longer allowed. This is now left to the definition of an enaction mechanism (i.e., events *entry_became_true* and *entry_became_false*). For a complete description we refer to the definition of attribute manipulations (see "5.5.2.5 Updating Attribute Values").

These modifications of the language definition were made both to increase readability and to improve expressiveness. Now one is able to specify a boolean expression which guards the related assignment statement. Thus, one is able to describe different reactions on the same event, depending on the actual project status.

4. New Types of Terminal Symbols

To get a homogenous typing philosophy of the elements in MVP-L, we introduced the following standard types to ease type checking of expressions and assignments and to unify the language constructs.

Firstly, MVP-L now provides the type *string*. Textual representations can be stored in attributes and compared for equality and non-equality.

Secondly, *real*-numbers are now included. This is needed for computing the exact value of model attributes (e.g., Software Science Metrics [10]).

Thirdly, a type *symbol* is introduced for representing user-specified values. These are needed, for example, for enumerating the status of a model.

5. Characters

Characters are now classified as expressed in the syntax. We distinguish between digits, upper and lower case letters, special characters, and characters which appear only in identifiers.

6. Expressions

Assignment statements were changed corresponding to the new types described. Thus, one can assign integer, real, symbol, string and boolean values to attributes.

7. Boolean Operator **xor**

To increase the readability of criteria the boolean operator **xor** is included in MVP-L. Often one must model that one or the other status is relevant, in the sense of exclusive states. Prior this could be expressed by using the boolean operator **or**. Now this fact is described by using **xor** which is closer to the semantics of such situations.

8. Parenthesizing of Object Composition

Because the "&", "|", and **exp** symbols used in object compositions are not real operands (i.e., they may not appear in expressions) the priority was not declared. Therefore, ambiguous composition of processes and products was possible. To avoid misinterpretation of such aggregated processes or products, parenthesizing was changed in this version of MVP-L.

Earlier versions of MVP-L:

```
object_relations
    ( object_1 & object_2 exp exponent_value )
```

Now:

```
object_relations
    ( object_1 & ( object_2 exp exponent_value ) );
```

or

```
object_relations
    ( ( object_1 & object_2 ) exp exponent_value );
```

9. Terminator

In some cases the separator ";" has changed to a terminator, i.e. every line of a list ends with a semicolon, even the last. This was introduced because practical work requires cutting and pasting of lines, and thus a consistent style of all lines should be provided.

10. Terminating Period

Models are no longer terminated by a period. This reduces redundancy because the end of a model is clearly stated by the keyword **end** followed by model's identifier.

11. Comments

The location of comments is now restricted. In the syntax it is explicitly described where comments are allowed in a MVP-L description. This was introduced to be able to generate the original text from an internal representation.

12. Syntax Modifications

One should not be confused when comparing both versions of the syntax. EBNF style notation is now introduced and some rules were restructured to increase understandability of the language's syntax.

The following changes have been made with respect to the version of MVP-L described in [8].

13. Imports in Product, Process, and Resource Attribute Models

The import clause is no longer part of the attribute models for products, processes, and resources. Because we decided that an instantiation parameter's type need not appear in the import specification of an attribute, this clause is empty for all attribute models.

14. Invariant for Process Models

Entry- and exit-criteria are only checked at the beginning or the end of a process activation. Modeling real-world software processes has shown the need for an invariant which is checked regular during the process lifetime. This invariant should remain true over the whole activation time of a process instance. In contrast to entry-criteria, which may turn to false after process activation, invariants document valid process states. One example for process invariants is to check whether the planned time for this process expired.

15. Filter for Tool Calls

Return values of tool calls were previously restricted to return type *real*. To allow a more flexible mechanism for passing attribute values to the process enaction mechanism, any basic type is allowed as the return value. However, only a subset of values may make sense. We introduced an extension (**in**-clause) to permit the range of values to be restricted.

16. Selective Event Mechanism

In attribute models one is not able to distinguish between different originators (process instances) of events. In some cases a process might be interested only in changing a subset of all product's

attributes. Therefore we introduced a filter for attributes of consumed and produced products within process definitions.

17. Consume-Produce Event

In order to allow adequate reactions of attributes to process consumption or production, we extended the list of standard events to be consistent with the three kinds of product flow relationships (i.e., consume, produce, and consume_produce). The consume_produce event was omitted in previous versions.

18. Request

A new concept has been introduced called **request** (rules 119 and 120). Similar to **call**, it is a hook into the real world that specifies the collection of metrics. A **request** is allowed only as the right hand side of an assignment statement. In contrast to **call**, it is not allowed to appear within expressions. There is exactly one argument of an request, which is a string appearing as message to the user. The user has to specify an attribute value, which is set after the request has been terminated.

19. Precedence of Operators

The precedence of operators has changed to C-like style. The former Pascal-like precedence was confusing to users of MVP-L (e.g., **and** had a higher priority than <=).

9 Acknowledgements

Special thanks go to Leo Mark who made many valuable contributions to the design of an early version of MVP-L.

Jürgen Münch validated an earlier version of MVP-L very carefully when implementing an internal representation of MVP-L models.

Jens Stummhöfer extracted many of the consistency rules from a recent version of this language report.

Leif Kornstaedt modified all existing language tools according to the new language definition.

This version of the language report contains a lot of suggested improvements made by various users of MVP-L. We would like to thank especially Ulrike Becker, Frank Diekhoff, Sven Gäßner, Sigrid Goldmann, Wolfram Petsch, and Barbara Swain for providing valuable comments on the language and its documentation.

10 References

[1] Victor R. Basili, "Software Development: A Paradigm for the Future", Proceedings of 13th Annual International Computer Software & Applications Conference, Orlando, FL, September 1989.

[2] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach, "Goal Question Metric Paradigm", Encyclopedia of Software Engineering, John C. Marciniak (Ed.), John Wiley & Sons, Volume I, 1994, pages 528 - 532.

[3] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach, "Experience Factory", Encyclopedia of Software Engineering, John C. Marciniak (Ed.), John Wiley & Sons, Volume I, 1994, pages 469 - 476.

[4] Victor R. Basili and H. Dieter Rombach, "Tailoring the Software Process to Project Goals and Environments," Proc. of the 9th International Conference on Software Engineering, Monterey, CA, March/April 1987, pp. 345-357.

[5] Victor R. Basili and H. Dieter Rombach "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, vol. SE-14, no. 6, June 1988, pp. 758-773.

[6] Ulrike Becker, "User Interface for the Software Process Modeling Formalism MVP-L", Projektarbeit, FB Informatik, Universität Kaiserslautern, 1994 (in German).

[7] Alfred Bröckers, "Process-Based Software Risk Assessment", Proceedings of the 4th European Workshop on Software Process Technology (EWSPT'95), Nordwijkerhout, Netherlands, April 1995.

[8] Alfred Bröckers, Christopher M. Lott, H. Dieter Rombach, and Martin Verlage, "MVP-L Language Report", Interner Bericht 229/92, Universität Kaiserslautern, December 1992.

[9] Mark Dowson and Christer Fernström, "Towards Requirements for Enactment Mechanisms", Proceedings 4th European Workshop on Software Process Technology, Grenoble, France, February 1994, Springer LNCS 772, 1994.

[10] A. Fitzsimmons, and T. Love, "A Review and Evaluation of Software Science", ACM Computing Surveys, vol. 10, no. 1, March 1978.

[11] Sven Gäßner, "Transformation of Software Process Models into the Internal Representation MVP-L-i", Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern, July 1993 (in German).

[12] Frank E. McGarry, "Results of 15 Years of Measurement in the SEL", Proceedings of 15th Annual Software Engineering Workshop, NASA's Goddard Space Flight Center, Greenbelt, MD, November 1990.

[13] David Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, vol.8, no.3, June 1987, pp.231-274.

[14] G. Kaiser, N.S. Barghouti and M.H. Sokolsky, "Preliminary Experience with Process Modeling in the Marvel Software Development Environment Kernel", Proceedings of 23rd Annual Hawaii International Conference on System Sciences, vol.II, January 1990, pp. 131-140.

[15] Marc I. Kellner and Gregory A. Hansen, "Software Process Modeling: A Case Study", Proceedings of 22nd Annual Hawaii International Conference on System Sciences, vol.II - Software Track, 1989, pp. 175-188.

[16] Marc I. Kellner and H. Dieter Rombach, "Comparisons of Software Process Descriptions", Proceedings of 6th International Software Process Workshop, Hakodate, Japan, ACM, October 1990.

[17] Marc I. Kellner et al., "Software Process Modeling Example Problem", Proceedings of the 6th International Software Process Workshop, October 1990, IEEE CS Press 1991.

[18] C.D. Klingler, M. Neviaser, A. Marmor-Squires, C.M. Lott, H.D. Rombach, "A Case Study In Process Representation Using MVP-L", Proceedings IEEE COMPASS, June 1992.

[19] Christopher M. Lott and H. Dieter Rombach, "A MVP-L Solution for the Software Process Modeling Problem", Proceedings of 6th International Software Process Workshop, Hakodate, Japan, October 1990.

[20] Christopher M. Lott and H. Dieter Rombach "Measurement-Based Guidance Using Explicit Project Plans", Journal of Information and Software Technology Vol. 35, No. 6/7 June/July 1993.

[21] Jürgen Münch, "Design and Implementation of an Object-Oriented Intermediate Representation for MVP-L", Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern, March 1993 (in German).

[22] Leon Osterweil, "Software Processes are Software Too", Proceedings of Ninth International Conference Conference on Software Engineering, Monterey, CA, March/April 1987, pp. 2-13.

[23] Dewayne E. Perry and Gail E. Kaiser, "Models of Software Development Environments", IEEE Transactions on Software Engineering, vol. SE-17, no. 3, March 1991, pp. 283-295.

[24] Bernd Peuschel and Wilhelm Schaefer, "ISPW-6 Exercise Solution", 6th International Software Process Workshop, Hakodate, Japan, October 1990.

[25] Proceedings of the 8th International Software Process Workshops. Wilhelm Schäfer (Editor), Dagstuhl Castle, Wadern, Germany, IEEE CS Press 1993.

[26] H. Dieter Rombach, "MVP-L: A Language for Process Modeling in-the-large", Technical Report UMIACS-TR-91-96, University of Maryland, 1991

[27] H. Dieter Rombach and Leo Mark, "Software Process & Product Specifications: A Basis for Generating Customized SE Information Bases," Proceedings of the 22th Hawaiian International Conference on Systems and Software, Hawaii, January 1989.

[28] H. Dieter Rombach and Bradford T. Ulery, "Establishing a Measurement-Based Maintenance Improvement Program: Lessons Learned in the SEL," IEEE Conference on Software Maintenance-1989 (CSM-89), Miami Beach, Florida, October 1989.

[29] H. Dieter Rombach and Martin Verlage, "How to Assess a Software Process Modeling Formalism From a Project Member's Point of View", Proceedings 2nd International Conference on the Software Process, Berlin, Germany, February 1993.

[30] H. Dieter Rombach and Martin Verlage, "Directions in Software Process Research", in Marvin V. Zelkowitz (Ed.), Advances in Computers, Vol. 41, Academic Press, Boston, MA, 1995.

[31] Jens Stummhöfer, "Consistency Rules for MVP-Models", Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern, October 1993 (in German).

[32] Enno Tolzmann, "Development of a Process Engine for MVP-L", Master Thesis, Fachbereich Informatik, Universität Kaiserslautern, January 1994 (in German).

[33] Niklaus Wirth, "Programming in MODULA-2", Springer Verlag, 1983.

Suggestions for improvement of this language report are welcome. Comments and questions should be directed to:

Martin Verlage
Arbeitsgruppe Software Engineering
Fachbereich Informatik
Universität Kaiserslautern
D - 67653 Kaiserslautern
Germany

Phone: +49 631 205 3337
Fax: +49 631 205 3331
Email: verlage@informatik.uni-kl.de

Appendix A

A small example of an MVP-L project description is given. The intention is not to provide a complete sample process model for software development projects, but to illustrate the use of some MVP-L concepts.

```
project_plan Design_project_1 is
    imports
        process_model High_level_design, Low_level_design;
        product_model Requirements_document, High_level_design_document,
            Low_level_design_document;
        resource_model Designer;
    objects
        requirements: Requirements_document('complete');
        high_level_design_doc: High_level_design_document('non_existent');
        low_level_design_doc: Low_level_design_document('non_existent');
        high_level_design: High_level_design(0, 1000);
        low_level_design: Low_level_design(0, 1000);
        d1, d2, d3, d4: Designer(0);
    object_relations
        high_level_design(req_doc => requirements,
                          hl_des_doc => high_level_design_doc, des1 => d1);
        low_level_design(hl_des_doc => high_level_design_doc,
                          ll_des_doc => low_level_design_doc, des1 => d2,
                          des2 => d3, des3 => d4);
end project_plan Design_project_1
```

Example 1: Design_project_1

The project plan in Example 1 instantiates a couple of objects (two processes, three products, and four objects representing project members). The total effort scheduled is 2000 units (e.g. hours) which is spent on both processes (see instantiation parameters of process *high_level_design* and process *low_level_design*).

```
product_model Requirements_document(status_0: Product_status) is
    product_interface
        imports
            product_attribute_model Product_status;
        exports
            status: Product_status := status_0;
    end product_interface

    product_body
        implementation
```

```
            -- This document contains the customer's and developer's
            -- requirements with respect to the overall software
            -- system.
        end product_body
    end product_model Requirements_document
```

Example 2: Requirements_document

Example 2 shows an elementary product model. All instances of that model have an attribute *status* which captures the actual state of the product representation.

```
product_model High_level_design_document(status_0: Product_status) is
    product_interface
        imports
            product_attribute_model Product_status;
        exports
            status: Product_status := status_0;
end product_interface

product_body
    implementation

end product_body
end product_model High_level_design_document
```

Example 3: High_level_design_document

```
product_model Low_level_design_document(status_0: Product_status) is
    product_interface
        imports
            product_attribute_model Product_status;
        exports
            status: Product_status := status_0;
    end product_interface

    product_body
        implementation
        -- Elementary product to represent detailed design of the software.
    end product_body
end product_model Low_level_design_document
```

Example 4: Low_level_design_document

```
product_attribute_model Product_status() is
    attribute_type
        ('non_existent', 'incomplete', 'complete');
    attribute_manipulation
        model_triggered
            when produce.start
```

```
                    and Product_status = 'non_existent' ->
                Product_status := 'incomplete';
            when produce.complete
                    and Product_status = 'incomplete' ->
                Product_status := 'complete';
            and others -> null;
end product_attribute_model Product_status
```

Example 5: Product_status

The product attribute model *Product_status* lists possible states of a product and specifies the state transitions. It should be noted that a product's state can also be changed during enactment by user interaction (e.g., putting the state back to 'non_existent'). This forced state transition is not specified in the overall project plan and invalidates the results of the analysis.

```
resource_model Designer(eff_0: Resource_effort) is
    resource_interface
        imports
            resource_attribute_model Resource_effort;
        exports
            effort: Resource_effort := eff_0;
    end resource_interface

    resource_body
        implementation
        -- An instance of this model represents a single member of the design team.
        -- Persons assuming the role of a designer must be qualified.
    end resource_body
end resource_model Designer
```

Example 6: Designer

The resource model in Example 6 describes the representation of a human agent. The attribute *effort* captures how much effort the resource has spent in this project.

```
resource_attribute_model Resource_effort() is
-- Effort data is collected each day the person is
-- logged in into the system.
    attribute_type
        integer;
    attribute_manipulation
        model_triggered
            when day_clock_event
                    and others ->
                Resource_effort := request ("Effort spent today?");
end resource_attribute_model Resource_effort
```

Example 7: Resource_effort

This attribute model specifies the effort spent by a single person. The data is collected by a request, which means an asynchronous communication between the process engine and human agent. The user fills in the data and sends the form back to the process engine, which in turn interprets the message and updates the attribute value.

```
process_model High_level_design(eff_0: Process_effort,
                                max_effort_0: Process_effort) is
    process_interface
        imports
            product_model Requirements_document, High_level_design_document;
            process_attribute_model Process_effort;
        exports
            effort: Process_effort := eff_0;
            max_effort: Process_effort := max_effort_0;
        product_flow
            consume
                req_doc: Requirements_document;
            produce
                hl_des_doc: High_level_design_document;
            consume_produce
        context
        entry_exit_criteria
            local_entry_criteria
                (req_doc.status = 'complete') and (hl_des_doc.status = 'non_existent'
                        or hl_des_doc.status = 'incomplete');
            global_entry_criteria
            local_invariant
                effort <= max_effort;
            global_invariant
            local_exit_criteria
                hl_des_doc.status = 'complete';
            global_exit_criteria
    end process_interface

    process_body
        implementation

    end process_body

    process_resources
        personnel_assignment
            imports
                resource_model Designer;
            objects
                des1: Designer;
        tool_assignment
    end process_resources
end process_model High_level_design
```

Example 8: High_level_design

```
        design_team: Design_group(0);
    object_relations
        design(req_doc => requirements_doc, des_doc => design_doc,
        designers => design_team);
end project_plan Design_project_2
```

Example 11: Design_project_2

The second project plan is an evolution of the scenario presented in Examples 1 - 12. The product flow, attribute relations, control flow, etc. have been validated, and the process engineer decides to package that knowledge into more abstract models and plans. Therefore, three new models are introduced (i.e., Design_document, Design, and Design_group). They now capture information which was described in the project plan above.

```
product_model Design_document(status_0: Product_status) is
    product_interface
        imports
            product_attribute_model Product_status;
        exports
            status: Product_status := status_0;
    end product_interface

    product_body
        refinement
            imports
                product_model High_level_design_document,
                              Low_level_design_document;
            objects
                hl_des_doc: High_level_design_document;
                ll_des_doc: Low_level_design_document;
            object_relations
                hl_des_doc & ll_des_doc;
            attribute_mappings
                status:
                    'non_existent' <-> hl_des_doc.status = 'non_existent'
                                       and ll_des_doc.status = 'non_existent';
                    'complete' <-> hl_des_doc.status = 'complete'
                                       and ll_des_doc.status = 'complete';
                    'incomplete' <-> others;
    end product_body
end product_model Design_document
```

Example 12: Design_document

The product model in Example 12 aggregates objects of the elementary product models shown in Examples 3 and 4. Thus the information about products captured in the project plan shown in Example 1 is a subset of the information captured in this product model. Additional information is provided in this model by the attribute *status* and the attribute mapping. It should be noted that the objects clause does not specify any values for instantiation parameters. The reason is that the instantiation of products is declared in process models (see Examples 8, 9, or 14). In product

models the objects are only used to specify allowed aggregates in the objects relations clause. An instantiation within a process model must match that specification.

```
resource_model Design_group(eff_0: Resource_effort) is

    resource_interface
        imports
            resource_attribute_model Resource_effort;
        exports
            effort: Resource_effort := eff_0;
    end resource_interface

    resource_body
        refinement
            imports
                resource_model Designer;
            objects
                d1, d2, d3, d4: Designer(0);
            object_relations
                (d1 & d2 & d3 & d4);
            attribute_mappings
                effort := d1.effort + d2.effort + d3.effort + d4.effort;
    end resource_body
end resource_model Design_group
```

Example 13: Design_group

The resource model shown in Example 13 aggregates four elementary resources of type *Designer*. Thus the information about resources captured in the project plan shown in Example 1 is a subset of the information captured in this resource model. Additional information is provided by the attribute *effort* which is the sum of all four elementary resource effort values.

```
process_model Design(eff_0: Process_effort, max_effort_0: Process_effort) is

    process_interface
        imports
            process_attribute_model Process_effort;
            product_model Requirements_document, Design_document;
        exports
            effort: Process_effort := eff_0;
            max_effort: Process_effort := max_effort_0;
        product_flow
            consume
                req_doc: Requirements_document;
            produce
                des_doc: Design_document;
            consume_produce
        context
        entry_exit_criteria
            local_entry_criteria
                (req_doc.status = 'complete') and (des_doc.status = 'non_existent'
```

```
                or des_doc.status = 'incomplete');
        global_entry_criteria
        local_invariant
            effort <= max_effort;
        global_invariant
        local_exit_criteria
            des_doc.status = 'complete';
        global_exit_criteria
end process_interface

process_body
    refinement
        imports
            product_model High_level_design_document,
                          Low_level_design_document;
            process_model High_level_design, Low_level_design;
        objects
            hl_des_doc: High_level_design_document('non_existent');
            ll_des_doc: Low_level_design_document('non_existent');
            hld: High_level_design(0, max_effort_0 / 2);
            lld: Low_level_design(0, max_effort_0 / 2);
        object_relations
            (hld & lld);
        interface_refinement
            des_doc = (hl_des_doc & ll_des_doc);
        interface_relations
            hld(hl_des_doc => hl_des_doc, req_doc => req_doc);
            lld(ll_des_doc => ll_des_doc, hl_des_doc => hl_des_doc);
        attribute_mappings
            effort := hld.effort + lld.effort;
            max_effort := hld.max_effort + lld.max_effort;
end process_body

process_resources
    personnel_assignment
        imports
            resource_model Design_group;
        objects
            designers: Design_group;
    tool_assignment
end process_resources
end process_model Design
```

Example 14: Design

The new process model Design (Example 14) aggregates the two elementary process models described in Examples 8 and 9. The interface relations clause exactly specifies the same relations between both processes as the project plan in Example 1. Additional information is provided by the attributes of the Design process and the personnel resources.