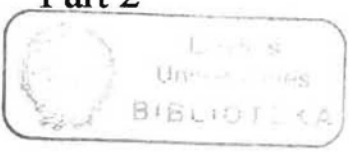
Audris Kalnins
Dr. Comp. Sci.

Automation of testing, specification languages and CASE tools

Habilitation Thesis

Collection of works

Part 2



Riga 1997

Also in this series

Functional Programming, Glasgow 1993
Proceedings of the 1993 Glasgow Workshop on Functional Programming, Ayr, Scotland, 5–7 July 1993
John T. O'Donnell and Kevin Hammond (Eds)

Z User Workshop, Cambridge 1994
Proceedings of the Eighth Z User Meeting, Cambridge, 29–30 June 1994
J.P. Bowen and J.A. Hall (Eds)

6th Refinement Workshop
Proceedings of the 6th Refinement Workshop, organised by BCS-FACS, London, 5–7 January 1994
David Till (Ed.)

Incompleteness and Uncertainty in Information Systems
Proceedings of the SOFTEKS Workshop on Incompleteness and Uncertainty in Information Systems, Concordia University, Montreal, Canada, 8–9 October 1993
V.S. Alagar, S. Bergler and F.Q. Dong (Eds)

Rough Sets, Fuzzy Sets and Knowledge Discovery
Proceedings of the International Workshop on Rough Sets and Knowledge Discovery (RSKD'93), Banff, Alberta, Canada, 12–15 October 1993
Wojciech P. Ziarko (Ed.)

Algebra of Communicating Processes
Proceeedings of ACP94, the First Workshop on the Algebra of Communicating Processes, Utrecht, The Netherlands, 16–17 May 1994
A. Ponse, C. Verhoef and S.F.M. van Vlijmen (Eds)

Interfaces to Database Systems (IDS94)
Proceedings of the Second International Workshop on Interfaces to Database Systems, Lancaster University, 13–15 July 1994
Pete Sawyer (Ed.)

Persistent Object Systems
Proceedings of the Sixth International Workshop on Persistent Object Systems, Tarascon, Provence, France, 5–9 September 1994
Malcolm Atkinson, David Maier and Véronique Benzaken (Eds)

Functional Programming, Glasgow 1994
Proceedings of the 1994 Glasgow Workshop on Functional Programming, Ayr, Scotland, 12–14 September 1994
Kevin Hammond, David N. Turner and Patrick M. Sansom (Eds)

East/West Database Workshop
Proceedings of the Second International East/West Database Workshop, Klagenfurt, Austria, 25–28 September 1994
J. Eder and L.A. Kalinichenko (Eds)

Asynchronous Digital Circuit Design
G. Birtwistle and A. Davis (Eds)

Neural Computation and Psychology
Proceedings of the 3rd Neural Computation and Psychology Workshop (NCPW3), Stirling, Scotland, 31 August – 2 September 1994
Leslie S. Smith and Peter J.B. Hancock (Eds)

Image Processing for Broadcast and Video Production
Proceedings of the European Workshop on Combined Real and Synthetic Image Processing for Broadcast and Video Production, Hamburg, 23–24 November 1994
Yakup Paker and Sylvia Wilbur (Eds)

Recent Advances in Temporal Databases
Proceedings of the International Workshop on Temporal Databases, Zurich, Switzerland, 17–18 September 1995
James Clifford and Alexander Tuzhilin (Eds)

Structures in Concurrency Theory
Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT), Berlin, 11–13 May 1995
Jörg Desel (Ed.)

Active and Real-Time Database Systems (ARTDB–95)
Proceedings of the First International Workshop on Active and Real-Time Database Systems, Skövde, Sweden, 9–11 June 1995
Mikael Berndtsson and Jörgen Hansson (Eds)

Recent Advances in Temporal Databases
Proceedings of the International Workshop on Temporal Databases, Zurich, Switzerland, 17–18 September 1995
James Clifford and Alexander Tuzhilin (Eds)

continued on back page...

Johann Eder and Leonid A. Kalinichenko (Eds)

Advances in Databases and Information Systems

Proceedings of the Second International Workshop on Advances in Databases and Information Systems (ADBIS'95), Moscow, 27–30 June 1995

Published in collaboration with the British Computer Society

Springer

Towards Integrated Computer Aided Systems and Software Engineering Tool for Information Systems Design*

Jānis Bārzdiņš, Ilona Etmane, Audris Kalniņš, Kārlis Podnieks
Institute of Mathematics and Computer Science
The University of Latvia
Rīga, Latvia

Abstract

The paper starts with a brief overview of the current situation in the world of CASE tools for information systems. Then there follows the outline of the basic ideas and principles of integrated CASE tool GRADE. The most outstanding characteristics of GRADE are that the tool is based on a unified specification language GRAPES and that it supports all information system development phases including analysis, requirements specification, design and implementation.

1 Introduction

It is a generally accepted view that complicated software systems including information systems can be built only using advanced CASE tools (see, e.g. [5, 6, 13]). The aim of this paper is to describe the basic ideas of the integrated CASE tool GRADE which is meant to support the building of complicated information systems.

But before we start to outline the basic principles of GRADE we want to characterize briefly the situation in the world of CASE tools for information systems.

It is generally accepted that development of complicated systems contains the following phases: analysis, requirements specification, design and implementation. The first important characteristic of a CASE tool is **the set of phases covered by the tool**. A typical situation is that most of well known CASE tools (Teamwork from Cadre Technologies, Oracle Case from Oracle Corp., NEW from Software AG, System Architect from Popkin Software etc.) cover only some of the development phases, most frequently, analysis only, or design only, or implementation only. In the contrast, GRADE is oriented towards more or less effective covering of all these phases and towards a seamless transition from one phase to the next one (like IEF from Texas Instruments and ADW from KnowledgeWare).

The second significant characteristic of a CASE tool is what **aspects of a system can be modeled by the tool**. By modeling a system aspect we understand more or less precise and formal description of this aspect. One

*This work was supported by Software House Riga and Infologistik GmbH, Munich

of the most popular approaches is to reduce the system modeling to its data modeling.

By data modeling one understands usually the building of the so-called conceptual data model in the form of entity-relationship diagram. In this case the software design and implementation is completely based on this data model. This approach is appropriate (and even has advantages) for small and medium size systems. This is due to the fact that the data model alone covers nearly all design needs for systems of this range and data modeling is a well-examined area, in addition.

However, things are completely different in the area of large systems. It is even impossible to understand a large system thoroughly without full-fledged modeling of all system aspects including general statical structure of the system, interfaces between components, data flows, control flows etc.

From this point of view, GRADE is a system which supports comprehensive modeling of very wide set of system aspects. More precisely, GRADE has facilities to model:

- the organization structure of a system,
- the so-called business processes performed by the system,
- the interfaces between system components,
- functions performed by separate components.

Data modeling (including ER models) is also supported by GRADE, but typically it should be used later, in the design stage. The basic paradigm of GRADE approach is that modeling starts not with data modeling, but with interface modeling, that is, with precise description of what information enters the system from outside and what information flows between separate components of the system. Data model appears in GRADE only afterwards, as the result of data flow modeling.

Further, CASE tools are also classified according to use of advanced specification languages with a fixed syntax and semantics or facilities for simple capturing of information in the form of tables as well as informal or semiformal diagrams. GRADE is completely based on a unified specification language GRAPES. The GRAPES language is founded on a graphical Siemens specification language GRAPES-86 [11]. In the framework of the GRADE project this language has got significant development and is extended by:

- 4GL level implementation facilities (GRAPES/4GL),
- special business modeling facilities (GRAPES/BM).

GRAPES-86 contains advanced facilities for describing system structure and interfaces of its components (communication diagrams, interface tables). According to GRAPES approach any modeled system is split into subsystems which communicate only via messages. Therefore it can be regarded that GRAPES execution semantics relies on the so called parallel communicating finite state machines model. GRAPES facilitates also a precise description of the logical structure of messages (data types being defined in data diagrams).

One more characteristic of CASE tools is how early in the system development process we get an **executable model** which can be demonstrated to the

customer, i.e., to what degree the **prototyping** is supported. A special feature of GRADE is that it supports executable prototypes in very early development stages and, again it is due to the fact that system models are being built in GRAPES which is an executable language. Another aspect of GRAPES is that its modeling features can easily evolve into programming ones, thus supporting seemless transition from system model to its target implementation.

Yet another aspect of a CASE tool is the gap between design and implementation. The world of tools is currently dominated by the approach where software design and implementation stages are separated. The design stage concludes (in the best case) only with a software specification, which is transferred further to manual implementation in the target environment (supported at best by environment-specific lower CASE tools). This approach has two serious drawbacks:

- the software implementation is only loosely linked to its specification, therefore series of software modifications, never reflected in its specification, occur during the maintenance,
- surplus costs are required for transforming the specifications into languages used by lower CASE tools.

In principle it is impossible to avoid completely the gap between modeling, design and implementation. However, this gap can be significantly reduced if the CASE tool is based on a unified specification language, as it is in the case of GRADE. In this case a system specification is refined further with each phase of the development process until finally it evolves into a formal system description, which can be compiled to the appropriate target environment. As it is well known, this is the approach used in telecommunication system area where SDL language is used as both specification and implementation language (see [12, 15]). GRADE with its GRAPES specification language is some attempt to use the same approach in the area of information systems, though the situation is much more complicated in this area. Yet another aspect where one benefits from the consistent use of one language is that the system specification may serve as a complete and correct documentation for the system implementation.

2 Basic components of GRADE toolset

GRADE toolset contains a vast number of different components which are closely coupled together. These components can be grouped into three large groups:

- Registrator,
- Business modeling components,
- Design and implementation components.

The Registrator component (designed by U.Sukovskis) is meant for initial capturing of information during system analysis phase. Registrator supports quick and easy information entry during interviews. All system objects are classified into active objects (performers), passive objects (messages and

stored data) and activities (functions), with simple predefined relations between them. The raw data afterwards can be automatically transformed into initial GRAPES models.

In a certain sense the Registrator ensures the functions of Repository in GRADE. In this paper we will not concentrate on the Registrator, but the main attention will be devoted to Business modeling and Design and implementation components, in the development of which the authors of this paper have participated essentially.

Business modeling and Design components will be described in next sections.

3 Business modeling components

Business modeling components are aimed at two goals:

- to define the organisation structure of an enterprise,
- to define business functions to be performed.

The GRAPES sublanguage used in this component is called GRAPES-BM. It is supported by modeling and simulation in GRADE toolset.

The **organisation structure** is defined using the following predefined entity classes:

- organisation unit,
- position,
- resource (equipment).

Each of the entities may be either single or multiple (representing a group of similar entity instances). A number of predefined relationships are introduced:

- consists of,
- has instance,
- uses.

Natural attribute sets for all entity classes are also predefined.

The organisation structure facilities cover the possibilities of OMT [14] to a certain degree, since the predefined classes and relationships encompass a large part of a general enterprise model to be described in OMT (though arbitrary classes and relationships are sometimes necessary).

The graphic editors in GRADE support an easy entry and modification of the organisation structure, an easy-readable tree-like information (ORG diagram) representation is used.

The **business functions** (called **business processes**) are defined using a special graphical sublanguage. To a certain degree this language is borrowed from [1], though it is significantly extended in the framework of GRADE project. The basic element of this language is the so called Task Communication Diagram (TCD). This diagram describes how a business process is split

into separate tasks, the intended sequence of these tasks as well as information and causuality links between these tasks. The links are represented via events.

Each task in TCD is associated with its triggering condition. The triggering condition is a boolean expression on possible incoming events of the task. An incoming event may be the reception from either of control from another task, or of a message carrying some data to be processed by the task. An incoming event may also be a timer, e.g., "at 8.00 AM daily". Event properties themselves are described in a special Event Table (ET). As soon as triggering condition is true, a new instance of the task is started.

The details of a task are described by the following sections of task description:

- Type of the task,
- Performer,
- Resource,
- Informal description,
- Objectives,
- Constraints,
- Execution mode,
- Attributes.

Types section specifies the type of the task; it is provided that tasks may have types posessing different sets of attributes.

The Performer and Resource section specify the necessary performers and resources (and their quantity) for the task to be executed. They both are boolean expressions on objects from the appropriate ORG diagram.

The Attributes section specifies the values of task attributes. These attributes may be either predefined, like Duration and Cost, or user defined. The attribute values may be defined as constants or expressions on other attributes, or data from incoming events, thus vital data dependencies also may be defined.

The other sections are more or less informal.

When a task instance execution terminates, the specified output events (messages) are generated. If necessary, the data contents of these messages may also be specified. Tasks may have also decisions, in order to define which output events are to be generated, according to the action results of the task. Decisions may be either informal (probability based), or formal, described by boolean expressions on task data.

Tasks in a TCD diagram should represent not only activities to be performed by information system under construction, but also all manually performed tasks. Namely this feature is characterized by Execution mode section.

As it is common is such systems, TCD diagrams support also multilevel task structuring. Top level tasks define the main business functions of an enterprise, and they are gradually refined into smaller tasks via subsequent TCD diagrams, until we reach the elementary task level. In the current version of GRAPES-BM the complete specification of an elementary task is informal (via sections

Informal description, Objectives, Constraints, Execution mode). In future versions of GRADE it is planned to introduce also formal specifications of such tasks in rule-based form. This approach, though in a slightly more theoretical manner, is outlined in [2]. Currently the formal task description elements (triggering, performers, attributes, decisions) yield an abstraction level sufficient for evaluating (via simulation) the overall system performance on time/cost basis and the necessary resources.

Historically the GRAPES-BM language has been inspired by Message Sequence Charts used in SDL [4] to describe behavior scenarios. GRAPES-BM significantly advances this idea branching structured scenarios. GRAPES-BM bears also some similarity to event-process-chain model [3]. On the other hand, GRAPES-BM facilities cover the traditional data flow modeling and dynamic modeling present in OMT approach [14].

The tool support for business modeling is both modeling and simulation oriented. The first direction is supported by very user-friendly editors for ORG, TCD and the other auxiliary diagrams/tables, with repository based automatic prompting and automated information transfer from diagram to diagram. Various automatic layout styles for TCD are supported. Thus very fast information entry and high degree of information integrity (with no model data ever entered twice) is ensured. Explicit global model consistency check is also supported.

The dynamic simulation feature supports a wide range of numerical estimates on the same model built for modeling purposes. Default statistical results include various time and cost-related performance statistics for the whole model and its elements, including workloads for performers, queue length for tasks etc. User defined statistics is based on user defined task attributes. Results may be displayed in both tabular and business chart manner. Thus a lot of model performance tuning may be done at a business modeling stage. A sort of TCD diagram animation is also supported to make model behaviour easily observable.

A certain amount of information from business model can be transformed into initial design model. A more complete information transfer will be supported in the next version of GRADE.

4 Design components

These have been historically the first components which were present already in the first version of GRADE. The components are based on the language GRAPES/4GL which can be used, on the one hand, as a design specification language and, on the other hand, as an implementation language having all typical 4GL level features.

As it was described above, business model describes a system in terms of business tasks, where one business process is performed, as a rule, by several performers. Now, when we pass to design phase, the main interest is what activities are to be performed by one performer. Especially the interest is focused on performers which are components of the information system under development. The behavior of such a performer may be obtained as a sum of all tasks where the performer participates.

According to GRADE methodology, the design phase starts with structural design. During the structural design the main GRAPES language feature to

be used is Communication Diagram (CD). GRAPES CD diagrams bear some similarity to block diagrams in SDL [4] and, in fact, are inherited from there. CD diagrams, in contrast to TCD diagrams used in GRAPES-BM, represent the splitting of a system into separate objects (which actually correspond to performers in BM representation) and communications between these objects by means of so-called communication paths. Each communication path is associated with its Interface Table (IT) which describes the data structure of messages sent along this path.

Thus by means of CD diagrams hierarchical decomposition of a system into its subsystems and then into lower levels is easily described until elementary objects are reached. To facilitate the description of message passing between several hierarchy levels, the so-called "channel concept" (which allows one to define the actual message sender/receiver freely and thus to build several IT's simultaneously) is used.

For complicated systems, according to GRADE methodology, the design is first performed at logical level. It means that only the data structure of a message is defined, but not the means of physical transferring this message (in most cases, the message will be transferred via screen forms). Thus the hierarchical decomposition, starting from top level objects of a system (departments, management, warehouse, etc.) ends with the lowest level (elementary) objects which are further refined by process diagrams. Thus, the next key element of GRAPES/4GL language is Process Diagram (PD). Process diagram describes in a graphic (and therefore, easy readable) form the behavior of a separate elementary object. The main components of a process diagram at this stage are message waiting/sending, which makes decisions upon message contents, and elementary data processing.

In addition, access to data bases may also be described at this level. It should be noted that in parallel with system decomposition GRADE supports also data design, and the conceptual data model should also be designed in the form of an extended entity-relationship model (ER diagram), the necessary data types are defined in a graphical form in DD diagrams. Therefore logical data manipulation aspects may also be designed at this level using advanced 4GL style data manipulation facilities referencing directly the components of the ER model.

The tool support of design components again consists of advanced graphical editors for all diagram types. The key feature of all these editors is high quality automatic layout of diagram elements, which may be easily combined with manual layout for some diagram parts. The second feature, already mentioned in BM support, is automatic prompting and consistency support. The prompting is crucial in efficient use of a PD editor where GRAPES language syntax has more textual elements, which could be otherwise difficult to remember.

The other most important tool at this stage is prototyper which ensures the model execution. When a logical design model (consisting of CD. IT, DD, ER, PD diagrams) is built, it can be executed in order to make some dynamic validation, to demonstrate it to the customer, and so on.

The logical design phase (which may be skipped for smaller systems) is followed by physical design phase. The same above mentioned set of diagrams is used in this phase. In addition to this, new types of diagrams - screen and report forms are also used to define the real user interfaces of the system. Starting from version 2.1, standard Windows GUI forms may be defined, containing all

traditional elements and facilities. In accordance with form design additional types of statements - 4GL style screen Input/Output statements are also used in PD diagrams, in order to manipulate these forms.

Now the prototyper may be used also to prototype the user interfaces of the system, in order to evaluate (by customer) real input/output forms, their outlook, ergonomics and so on. It should be noted that in principle screen interfaces may be designed very early in the design process, even when there are no real data. Thus various types of prototyping may be freely mixed up, since the same language and tool is used for all of them. Thus the methodology can be adapted to any specific user demands, and, if it is required so, the design and prototyping may be started from user interfaces and even the form dynamics may be prototyped in that case (with fixed data, as a rule).

When the system design model is validated thoroughly via prototyping, its implementation starts. Again the same GRAPES/4GL language is used, but now in its full scale as a programming language. Namely, all advanced ER-based data manipulation facilities are used. All input/output dynamics details are described in the same way, in order to define all exceptional situations, data validations and so on. The prototyper again is used, but in a role of language debugger, with advanced debugging facilities. Then the validated implementation model is passed to code generator which generates actual code for one of the selected target environments. Currently, in version 2.0, Informix database environment is supported either for MS DOS, or UNIX. In the next version 2.1 the Oracle environment will be supported, with the generated application running as client in MS Windows. For all environments, the GRADE code generator generates C code with embedded SQL statements, which is automatically compiled into ready-to-use applications. No code maintenance at C level is necessary. Sufficient efficiency of the generated code is guaranteed, some manual optimization hints may be added to ER model definition before the code generation.

5 Current state of GRADE tool

Since 1993 version 1.0 of GRADE is being distributed by Siemens-Nixdorf [7, 8] This version does not support Business modeling. Since February 1995 version 2.0 is being distributed [9, 10], this version supports Business modeling and multiuser development mode. Version 2.1 is in preparation (planned delivery December 1995). This version will contain Graphical User Interface (GUI) screen forms, advanced Business modeling facilities and extended data dictionary facilities. The tool performance will also be significantly improved, especially for multiuser network environment.

Acknowledgements

The GRADE toolset is the result of intensive labour of about 30 developers' team during several years. The authors of this paper wish to use the opportunity to thank all their collegues for mutual understanding and assistance during the project development. They also wish to gratefully acknowledge Software House Riga and Infologistik for the financial support of the project.

References

[1] A. Aue and M. Breu. Distributed information systems: an advanced methodology. *IEEE Transactions on software engineering*, 20(8):594–605, 1994.

[2] J. Bārzdiņš, G. Bārzdiņš, and A. Kalniņš. Rule-based approach to business modeling. In *Proceedings of the SEKE95*, 1995.

[3] W. Brenner and G. Keller, editors. *Bussiness Reengineering mit Standardsoftware*. Campus Verlag, 1995.

[4] *CCIT. Message Sequence Charts.*, 1992. Recommendation Z.120.

[5] M. Chen and J.R. Normannn. A framework for integrated CASE. *IEEE Software*, 9(2):18–22, 1992.

[6] A. Fugetta. A classification of CASE technology. *IEEE Computer*, 26(12):25–38, 1993.

[7] *GRADE Version 1.0 Language Description*, 1993.

[8] *GRADE Version 1.0 User's Guide*, 1993.

[9] *GRADE Version 2.0 Language Description*, 1995.

[10] *GRADE Version 2.0 User's Guide*, 1995.

[11] G. Held, editor. *Sprachbeschreibung GRAPES: Syntax, Semantik und Grammatik von GRAPES-86*. Verlag Siemens AG, 1990.

[12] V. Klick, J. Patti, and M. Todd. Experience in the use of SDL/GR in the software development process. In *SDL91: Proceedings of the 5-th SDL forum*, pages 449–457. North-Holland, 1991.

[13] P. Loucopoulos and B. Theodoulidis. CASE methods and support tools. In P. Loucopoulos and R. Zicari, editors, *Conceptual modeling, databases and CASE: An integrated view of information systems development*, pages 373–388. John Willey & Sons, New York.

[14] J. Rumbough et al. *Object oriented modeling and design*. Prentice Hall, 1991.

[15] A. Zaim and F. Calikoglu. Using SDL in a commercially available wide area coverage trunking mobile radio system development. In *SDL93: Proceedings of the 6-th SDL forum*, pages 41–49. North-Holland, 1993.

Second International Baltic Workshop on

Databases and Information Systems

organised by

Institute of Cybernetics
Tallinn Technical University
CIDEC of the Estonian Universities

sponsored by

Estonian Informatics Fund
Baltic Fund of VLDB Endowment
Swedish Institute for Systems Development
The Baltic Institute of Finland

Hele-Mai Haav, Bernhard Thalheim (Eds.)

Databases and Information Systems

Proceedings of the
Second International Baltic Workshop
Tallinn, June 12-14, 1996

Volume 2: Technology Track

Business Modeling Language GRAPES-BM and Related CASE Tools

A.Kalnins, J.Barzdins, A.Auzins,
I.Etmane, A.Kalis, K.Podnieks, J.Tenteris, E.Vilums, A.Zarins

University of Latvia
Institute of Mathematics and Computer Science
Rainis Blvd. 29, Riga LV-1459, Latvia
and RITI, Skanstes Str. 13, Riga LV-1013, Latvia

Abstract

Business modeling language GRAPES-BM is a semiformal graphic language for modeling and simulation of complicated business systems (production processes, offices, information systems). GRAPES-BM relies on such basic concepts as task, event, performer, triggering condition, etc. and contains advanced facilities for describing system behaviour ("business process"). GRAPES-BM contains also advanced facilities for modeling the static structure of a system. CASE tools based on GRAPES-BM support graphic modeling and simulation.

1 Introduction

The term Business Modeling (BM) has become a buzzword during last few years. There is no unique definition of BM. Different people understand different things under this term. There is, however, something common to all these approaches. BM is closely related to another buzzword, namely, Business Process Reengineering (BPR), and constitutes the most well understood part of it. Any BM approach tries to present semiformal graphical means for describing behaviour and structure of complex business systems. This description is in the form of interrelated diagrams of various kinds. The main use of such description is to comprehend thoroughly and unambiguously such business systems.

In order to understand the behaviour of a system, it is necessary to understand activities within this system, causal links between these activities, their stimulus and results. In most cases the behaviour is being described by diagrams consisting of symbols (rectangles, bubbles etc.) representing activities, various connecting lines representing the links and, possibly, some auxiliary symbols.

The first such formalism was Data Flow Diagrams [1], which were introduced for other purposes and only sometimes are used for BM. A number of similar more or less specific formalisms followed; among them function dependency diagrams [2], event schemas [3], EPC diagrams [4. 5], Business process diagrams [6] etc. At the given moment none of the formalisms is universally accepted.

GRAPES-BM, described in this paper, is also a semiformal approach for describing behaviour and structure, using similar graphic notation as it basis. The main difference from all abovementioned languages is the level of formality. While in most of the existing approaches [2, 3, 4] the formality level is rather low, in GRAPES-BM it may be varying,

from very informal use up to a very formal, nearly program-like description of a business system.

The other important criterion is the possible tool support in the analysis of the described business system. Again the more formalized is the approach, the richer set of tools is available. In most cases, e.g., [5], the tool support reduces to simple consistency checking, reporting and some static evaluation, most frequently, finding the critical path in a weighted flow graph. The other tools offer dynamic prototyping and simulation as the main analysis method. Among the known BM systems Designer 2000 [6] should be mentioned.

GRAPES-BM with its support tool GRADE is mainly dynamic execution oriented. Even very informal GRAPES-BM models may be executed in a sense thus giving much deeper insight into system behaviour and its possible bottlenecks. On the other hand, for highly formalized models precise simulation of quantitative aspects is possible in a way close to specific simulation languages.

Now some words on history of GRAPES-BM. Development of GRAPES-BM started with A.Aue and M.Breu paper [7]. Afterwards M.Breu, A.Mraz, N.Richter, at al (European methodology and System Centre) have issued several preprints where the concept was developed further. On the basis of these works G.Barzdins, J.Barzdins and A.Kalnins created GRAPES-BM, version 2.0, which was implemented in the tool GRADE 2.0 [8, 9]. Usage of GRADE 2.0 revealed further development possibilities. As a result, a substantially new version of the language, called GRAPES-BM, version 3.0, was created. GRAPES-BM described in this paper corresponds to version 3.0. In the development of this language version and corresponding tool set (GRADE 3.0), besides the authors of this paper, the following people have made significant contribution (in alphabetic sequence):

D. Foerster (SNI - Germany), E. Knoener (SNI - Germany), C. Rositani (SNI - Italy), U. Sukovskis (RITI), A. Teilans (RITI), M. Weiss (SNI - Germany), U.O. Ziemelis (INFOLOGISTIK).

2 Goals of Business modeling in GRAPES-BM

First, let us be more specific towards what kind of universe of discourse GRAPES-BM is oriented. Classical system specification languages e.g. SADT, IEF[11] are mostly aimed to semiformal description of **Information systems (IS)** in their early development stages. On the contrary, BM approach is intended to describe a significantly wider class of systems. Typical examples are large organizations, complete enterprises, production systems. Only part of such systems is IT related, the other part is completely human related, like it is, e.g., in airline ticket reservation system. We follow the latest traditions and call such systems **Business Systems (BS)**. Then IS can be considered as a part of such BS. The main task of GRAPES-BM is to support convenient description of BS.

The requirements for the language are extremely contradictory:

- on the one hand, the language should be easy-to-read for anybody, including top managers
- on the other hand, it must be formal enough to support unambiguous interpreting by IT professionals and to permit dynamic execution and simulation for obtaining numeric evaluations.

GRAPES-BM seems to have succeeded in combining these two requirements. The formal and informal aspects of language are so naturally coupled, that even a very formal description may be understood quite intuitively (certainly, after some language training).

The formal goals of developing Business models in GRAPES-BM are to facilitate Business process reengineering by

- providing precise and readable at the same time business system description, as the basis for main reenginering decision making
- investigating alternative ways of behavior of the BS, using various prototyping and animation facilities
- simulation of the model to reveal possible bottlenecks and measures to avoid them.

It should be stressed that extension of a GRAPES-BM model to a simulatable one requires only adding some numeric attributes in the model already built.

If BS includes also an IS which should be reengineered, than relevant parts of the developed BM serve as a formal high level requirements specification for the new IS development. GRAPES-BM is well suited for this purpose. Certainly, GRAPES-BM should not be considered as a design language for IS, another GRAPES family language GRAPES/4GL [8] should be used there instead (see also [10]).

3 Main concepts of GRAPES-BM

Business modeling in GRAPES is based on two fundamental concepts: tasks and events.

3.1 Tasks

According to Websters dictionary **Task** is defined as "a piece of work". Any activity which is performed in a business system to be described is considered to be a task. Tasks may be very large - defining one basic activity of an enterprise and very small - like signing of a document. Large tasks are decomposed into chains of smaller ones using **Task Communication Diagrams (TCD)**. These diagrams are the basic ones in GRAPES-BM. Each task has its **name**. But it may have other formal attributes like

- performer of a task
- triggering condition
- duration.

Graphically (in TCD diagrams) tasks are represented by rounded rectangles. This rectangle shows the task name and its basic attributes, e.g., performer.

There are two types of tasks (examples are shown in Fig. 1):

- ordinary (transformation) tasks
- decision (branching) tasks.

Decision tasks have two or more named decision symbols attached to them. During execution of them one of the alternative outputs are chosen.

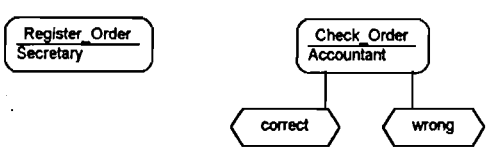


Fig. 1 Examples of tasks

It should be mentioned that concept of task is present in any business modeling approach, only the terminology is quite different. Tasks are called functions in function hierarchy and dependency diagrams [2], process steps in ORACLE process diagrams [6], operations in Martin's OOA event schemas [3] etc.

3.2 Events

The other fundamental concept of GRAPES-BM is event. Events represent anything that can happen in a business system. Events are also the other principal element of Task Communication diagrams. They are represented by arrows leading from one task to another.

There are several categories of events:

- **message events**

Events with category "message" correspond to objects produced by one task and transmitted to another. This concerns materials (e.g., paper, part of machine) and pure information (invoice, bill, report).

Message events always have **name** which is depicted next to the arrow.

Message events can carry information with them. The information is represented as datatype associated with the event. The association is described in **Event Table (ET)** - an object global for the whole business model.

- **control flows**

They express the fact that one task is completed and the next task may start. Control flows are represented as unnamed arrows.

- **timer events**

These are the only events not created by tasks. They appear in certain time moment from an abstract timer which is represented as a small clock and go to the task pointed by the arrow. Each timer event has a name. The exact definition of time moments for a timer is given in the Event table.

Fig. 2 shows examples of events as they are depicted in TCD diagram.

One task can produce more than one event at the output. Similarly, one task can have several input events arriving from different tasks.

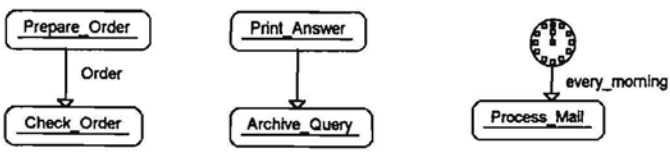


Fig. 2. Examples of events

3.3 Task details

The next fundamental concept of GRAPES-BM language is **triggering condition**. It is associated with a task, as one of essential its properties. Triggering condition specifies which combinations of input events are necessary to start the task. This condition is specified as a boolean formula containing ANDs and ORs on event names e.g., *Order AND Payment*.

The general event semantics principle in GRAPES-BM is that incoming events form FIFO queues in front of a task (separate queue for each event name). When triggering

condition becomes true, the task consumes the relevant set of events from its input queues and starts execution. In the simplest case the triggering condition may be reduced to simple AND (or even "&" sign), which means ANDing all possible input events (i.e., one from each queue). Control flows also are implicitly ANDed in this case (i.e., all of them must be present). Similarly, simple OR (or "|" sign) means that any one of input events (including control flows) is sufficient for triggering. If any of the required events is not present, the task waits for its arrival.

The next important part of a task is its **performer**. Performer specification consists of one or more performer names connected by AND and OR connectors. Performers may be organizational units, persons (positions) and equipment (resources). The available performers and their number are specified in the ORG diagram of the business model. The requested performers must be free before the task can really start, therefore the triggering condition is only the necessary condition for a task to start.

Duration of the task specifies the required execution time, e.g., task *Order_processing* in fig. 3 takes exactly 1 hour.

Fig. 3 shows an example of completely specified task in a TCD.

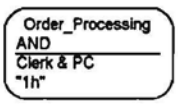


Fig. 3 Completely specified task

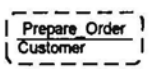


Fig. 4. External task

Once task has started, it performs its main activity, its "piece of work", which is not formally specified in GRAPES-BM. When the task is completed, it possibly takes one of its decisions and sends its output events.

More of task's formal and informal details may be described in its **Task Specification Diagram (TSD)**. In particular, extended informal description of a task may be given there.

In conclusion one more remark on tasks. In any behaviour description there are tasks which are not part of the business system under consideration, e.g., customer preparing an order. Such tasks are called **external tasks** in GRAPES-BM. They are represented using dashed lines for task symbol. Fig. 4 shows an example of external task:

3.4 Data manipulation

Data manipulati in GRAPES-BM on is described only at informal level. There are two special symbols for that purpose.

Data store stands for a persistent (independent from the current task) storage of data or materials. Typical use of data store is for existing databases in the IS part of a business system. In that case its contents can be described in detail by **Entity-Relationship (ER)** diagram, which also may be a part of a business model in GRAPES-BM. On the other hand, data store may also represent informally a stock of goods. Data stores have names in GRAPES-BM, and they are connected to tasks by lines called **Access Paths**.

Data Object is supposed to represent just one object, and with a shorter life time - just one business transaction. Data object again may represent a physical object or data object (global variable) at IS level. In the latter case its data type may also be specified (with type definition being given in a **Data Definition (DD)** diagram, which may also be a part of business model).

Fig. 5 gives an example of data store and data object.

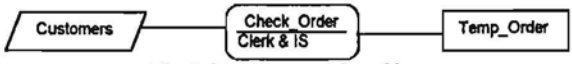


Fig. 5 Data store and data object

4 First Insight into Business Process

The main goal of business modeling is to describe both readably and concisely a business system behaviour. As it was already pointed out, the main sort of diagrams for this purpose is Task Communication Diagram (TCD), describing a behaviour of large task in terms of

- smaller tasks
- events
- data stores and data objects.

As a rule, from the informal point of view such a description represents a reasonable **business process** in a system. Therefore the concept of business process is the informal equivalent of the formal TCD concept.

Let us consider an example of a simple business system, namely, a small office providing consultations for customers. The office consists of a chief, a secretary and a PC. Fig. 6 presents a business process describing just one aspect of the office activities - processing of incoming mail and providing written answers to queries of customers. The office receives letters (written queries), secretary registers them and afterwards the chief and secretary together make the answer. The secretary uses PC to type and print the answer which is sent to customer at the end. The actions performed by customer and Information Source are external to the office and therefore are shown by dashed lines.

5 Precise Semantics of Business Process, Concept of Transaction

The example in the previous section could be understood and even analyzed quite intuitively.

However, GRAPES-BM has completely precise semantics defined. This semantics may be used for unambiguous manual validation of business models and for their execution by GRADE tools, i.e., simulation, prototyping, animation. It should be emphasized that fig. 6 constitutes a syntactically correct and executable TCD diagram (certainly, in the context of some definition diagrams to be discussed in section 7).

Timers in a TCD diagram are spontaneously active elements, i.e., they send their events to the appropriate tasks. These tasks are then triggered and afterwards they send their resulting events to other tasks. Thus the whole business process gets into motion. There may be as many concurrent instances of any task active as available performers permit it.

But there is one completely novel element added to this relatively straightforward semantics. This is the concept of **Transaction**.

Intuitively a business transaction is a chain of activities initiated by some external stimulus and ended at the moment when further events are beyond our scope of interest. In the example of Fig. 6 the transaction starts with the arrival of a new query from the customer. To be more formal, it starts from the moment when timer *Regularly* starts the external task *Send_Query*. The transaction is completed when *Answer* is sent to external task *Receive_Answer*.

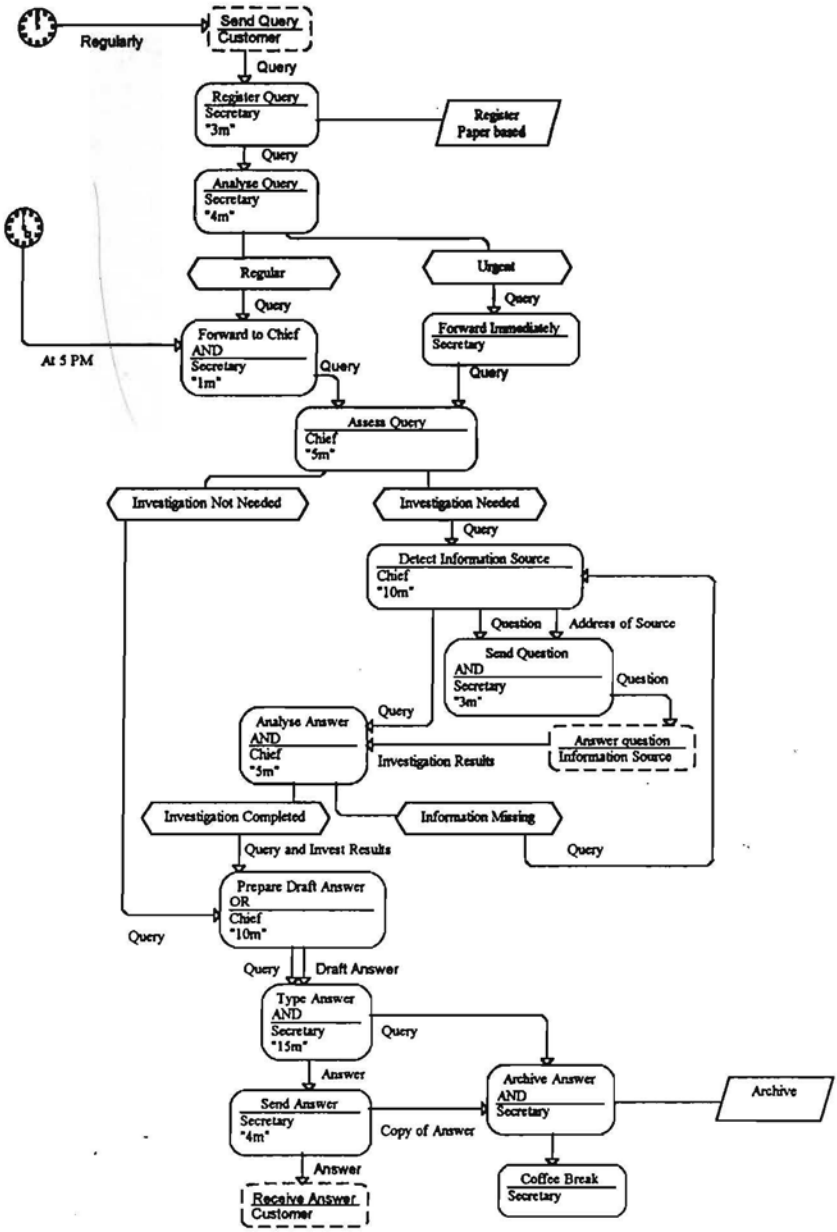


Fig. 6 Example of business process (TCD diagram)

The concept of transaction is fundamental in business modeling, since it helps to find out and analyze essential groups of activities inside a business system.

The main problem here is to find a simple formal definition of transaction which would coincide with the intuitive understanding in most cases.

In GRAPES-BM the following definition is used:

The **transaction starts** only when a task is started **only** by events coming from outside the business systems.

Two types of events in GRAPES-BM are defined as such "outsiders":

- timers, described already in the previous section
- spontaneous events. Any event may be made spontaneous in the TSD diagram by assigning generator definition to it. These generator definitions have the same syntax as timer definitions. Spontaneous events are used to hide away a timer in TCD diagram and to make the impression that the event comes right from an external task.

Thus, in fig. 6, the timer *Regularly* starts a transaction since the task *Send_Query* is triggered solely by it. But the timer *At_5_PM* starts no transaction, since *Forward_to_Chief* requires another (internal) event *Query* in order to be triggered.

The precise description of transaction behaviour is based on so called **Transaction Identifier (TID).** At the beginning of each transaction the starting event is given a unique TID. This number will be used throughout the transaction, all events and tasks in the corresponding task chain will be tagged by it.

There is no explicit use of TID. However, it participates implicitly in each triggering condition. AND condition will be true only if all incoming events have the same TID. Thus only matching groups of events belonging to the same transaction can trigger a task. In fig. 6 only those *Investigation_results* which correspond to the *Query* will trigger the task *Analyse_Answer*.

Transaction is completed when there are no more events in the model with the given TID. In some occasions default rules are insufficient. To cope with these situations the following options of tasks may be used:

- NOSTART for preventing an unwanted start of transaction
- START for explicit start of transaction
- END for explicit end
- NOTID for explicit stripping off the TID from an event.

6 Description of Organization Structure

So far the description of system behaviour in GRAPES-BM has been outlined. The other important business system aspect is structure description.

In GRAPES-BM this is done via **ORG** diagram. Fig. 7 shows the ORG diagram for the office example.

The example should be self-explanatory, since it strongly reminds traditional org-charts.

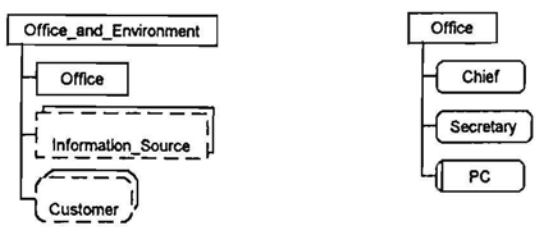


Fig. 7 Example of ORG diagram

More formally, ORG diagram may contain

- **organizational units** (company, department, group etc.)
- **positions** (chief, accountant, secretary, programmer etc.)
- **resources** (any kind of equipment, like car, PC etc.)

Any of the elements may be single or multiple, for multiple elements the number of available instances may be specified (otherwise unlimited number is available).

Organization structure is depicted as a tree (more precisely, as a set of trees) built from the abovementioned nodes. The edges of the tree represent:

- **contains** relationship between unit and its subunits, unit and its positions and resource and its components
- **owns** relationship between unit and resource and between position and resource.

The same line type is used for both relationships since the proper relationship can always be deduced from the context.

A leaf of a tree may be refined further by another tree.

Any of the organizational structure elements may have the following additional attributes:

- type (internal or external)
- competence list
- availability (as time interval)
- cost per hour
- efficiency level
- employee name (for single position only)

It should be remarked, that though GRAPES-BM is not an OO language, ORG diagram facilities permit one to depict a great deal of information typically found in OO models (e.g., OMT [12]), for example, subtyping may be represented via competence.

All ORG diagram elements have also precise formal semantics, which is taken into account when TCD diagrams with performer specifications in tasks are being executed.

7 Business model of System

The previous sections have given some insight into two most significant diagrams of GRAPES-BM - Business process (TCD diagram) and ORG diagram. Besides that, several types of diagrams have been simply named: ET (event table), TSD (Task Specification Diagram), ER and DD diagrams.

A complete Business model is a hierarchy of abovementioned (and some other) diagrams. The hierarchy itself is defined in a **model tree.** Fig. 8 shows the model tree for the office example. Model tree may be considered as a table of contents for the model.

The top line contains the elements global for the whole business model:

- ORG diagram
- ET table
- CMP table

ORG diagram has been discussed briefly in the previous section.

ET table has a row for definition of each event used in the business model. The most complicated are the timer definitions.GRAPES-BM provides formal means for timer definitions, e.g.,

AT_5_PM could be defined as *TIME("*.*.* 17:00"),*
Regularly as *REPETITION("1h:30m").*

Message events may have their data types specified (record or elementary types may be used). **Competence table (CMP)**is a supplement to ORG diagram, listing possible competences of ORG elements.

The main part of model tree is constituted by **primary tasks** (top-level TSD diagrams) and their refinements. There is only one primary task *Query_Processing* in the example. In general, primary tasks correspond to the main relatively independent functions of an enterprise. Each primary task normally is refined by its TCD diagram (business process), shown to the right of the corresponding TSD diagram.

The next level of refinement is defined by the set of subordinated TSD diagrams corresponding to all tasks mentioned in the TCD diagram.

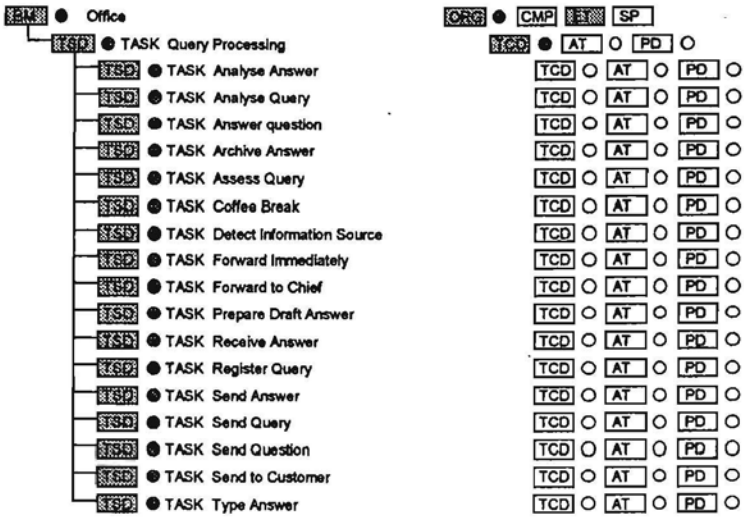


Fig. 8 Model tree

In the simplest case a TSD diagram contains the same formal attributes of task as those visible in the TCD plus extended informal description of the task. In addition, the task's interface to its environment is also visible in TSD via so called referenced tasks. Fig. 9 shows the TSD diagram for the task *Analyse_Query*. In general case, however, TSD diagram may contain significantly more information (briefly sketched in the next section) which has special value for simulatable models.

The Models with one primary task and one TCD diagram refining it are called **flat** models. However, in general case the situation is much more complicated. Each task in the TCD diagram may be further refined by its own TCD diagram (placed in the tree in the same line as the corresponding TSD diagram). The refinement is continued until we obtain the lowest level tasks which are called **elementary tasks**. For elementary tasks their formal and informal characteristics can only be specified in their TSDs. The choice of elementary tasks depends on the specific application of business modeling. The abovementioned language facilities show how traditional **structural refinement** is supported in GRAPES-BM.

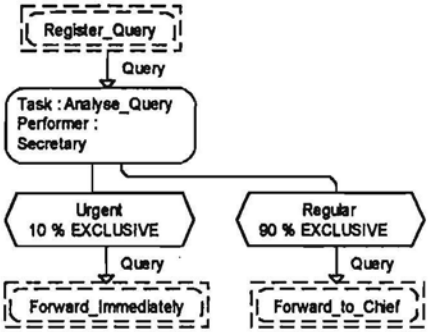


Fig. 9 Example of TSD diagram

One more type of diagrams to be mentioned is **attribute tables (ATR)**. They are global for the whole model, and there may be several named ATR tables. Each ATR table describes user-defined attributes for the given task type, which is equal to the ATR name.

Zero or more DD diagrams are also global for the whole model. They are placed in model tree above TSD diagrams.

8 Advanced Features of GRAPES- BM

The items discussed so far have been more or less related to semiformal business description and analysis.

However, GRAPES-BM permits to describe precise behaviour of business systems from the control point of view, including some data-related dependencies. This layer of GRAPES-BM actually constitutes a sort of process simulation language.

The basis of all these features is the assumption that events can carry data with them, and the data may be "processed" by tasks, used in decisions and transferred further to output events.

The following features are available:

- advanced triggering conditions, like
  - *Letter AND ALL Answer WHERE Letter.Id= Answer.Id*
  - *Letter AND <5> Comment*
- detailed description of task decisions
  - by their probabilities, in exclusive or nonexclusive manner
  - by precise formulas which may depend on data carried by triggering input events and on numeric attributes of the task
- task duration dependent on its input data
- formulas for setting values of user-defined task attributes (then the task must have one of the types defined by ATR tables of the model). Formulas may reference data of input events and other attributes
- formulas for setting data (record fields, as a rule) in output events of the task, with formulas referencing similar class of values. There is a special convention of data passing , namely, if an output event has the same name as an input event, data are passed without any formulas specified
- repetition factor for output events.

These described features may appear in TCD as well as in TSD diagrams. The "computational aspects " are taken into account only for elementary tasks.

The described features allow one to describe adequately various control structures present in business systems, like:

- iterative looping depending on event data
- centralized control depending on some global data
- time-out control of incoming events

The "programming " of such control structures is sufficiently simple. The methods used remind slightly those used in "programming " of Petri nets. Significant role here is played also by transaction concept. In most cases the precise control aspects may be simply added to models originally built for pure qualitative analysis.

9 Short Overview of GRADE Modeller Tools

The full support of GRAPES-BM language is included in the new version 3.0 of GRADE Modeller toolset.

The following components of GRADE are available:

- tree and repository management
- advanced graphical editors
- syntax analyzer
- GRAPES-BM language interpreter (BM-simulator)
- animator
- trace browser

A key element in GRAPES-BM support is the graphical editor set for all types of described diagrams. Editors make model development simple and attractive due to the following features:

- highly optimal automatic layout for all kinds of diagrams. Several styles for such layout may be defined. Automatic layout smoothly coexists with manual layout improvements for presentation purposes
- the relevant name and syntax construct prompting
- automatic transfer of the relevant information from one diagram to another. Though GRAPES-BM language requires some information duplication between TCD and TSD diagrams (for better readability), no data must be entered twice - the editors automatically transfer the data to the required direction
- automatic updating of tables (ET, CMP) during diagram construction
- automatic TCD templates when refining a task by a new TCD diagram level

For logically simple (but may be, large) business models the only diagram types to be explicitly built are ORG and TCD diagrams.

Though a lot of inter-diagram consistency requirements are ensured by editors, extensive analysis is still necessary. The diagnostic messages are shown via the same editors. For semiformal use of GRAPES-BM the syntax analyzer plays the role of diagram consistency checker. The other result of analyzer is the intermediate code of diagrams used for execution.

GRAPES-BM is built as an executable language and therefore BM-simulator plays a significant role in business model development. It has the following features:

- step mode with variable granularity for business model dynamic debugging and step-by-step exploration

- run mode for business model prototyping and simulation. The run mode is combined with pause and breakpoint features
- inspect facility for observing any elements of the current status of business model (active tasks, event queues, data contained in them, etc.)
- user-controlled automatic statistics gathering (for predefined statistical features of tasks, events and performers and for statistics of user-defined task attributes)
- interface to diagram animator

Animator is used for on-line animation of selected TCD diagrams. Active tasks (including the number of instances), the events passing along their routes and length of event queues are shown in these diagrams. The collected statistics can be viewed both in tabular and chart (EXCEL-like) form, using the trace browser component.

Since even quite informal BM models are executable as a rule, BM simulator serves as a powerful tool for model validation, step mode execution combined with animation allows one to find any unexpected behaviour of the model. On the other hand, normal animated run of a model is very helpful in general evaluation of the model and in finding unexpectedly long queues and other bottlenecks in the system.

Automatic statistics gathering supports easy simulation experiments with business models.

10 Conclusions

The business modeling language GRAPES-BM seems to have taken its stable place among other BM languages. The first pilot applications of GRAPES-BM (design process management in car industry, some banking applications, public utility management et al) have shown its feasibility for description of comparatively large business systems. The main novel feature seems to be the wide spectrum of applicability of the same models, from general informal evaluation of the current system to numeric experiments with it.

Future development directions of GRAPES-BM are now being discussed. One of such directions could be Rule-based approach [13]. but the problem is that the language must be kept simple enough in order to be understood by users not being IT professionals.

References

[1] DeMarco, T.: Structured Analysis and System specification, Prentice-Hall, 1979.

[2] Barker, R., Longman, C.: CASE*METHOD Function and Process Modeling, Addison-Wisley, 1992.

[3] Martin, J.,.Odell, J.: Object-Oriented Analysis&Design, Prentice-Hall, 1992.

[4] Keller, G., Nüttgens, M., Scheer, A.W.:. Semantische Prozessmodellierung auf der Basis Ereignisgesteuerter Prozessketten (EPK), in *Veroffentlichungen des Instituts fur Wirtschaftsinformatik*, v. 89, Saarbrucken, 1992.

[5] Brenner, W., Keller, G. (Eds): Business Reengineering mit Standartsoftware, Campus Verlag, Frankfurt, 1995.

[6] Designer-2000. A Guide to Process Modeling. Oracle Corp., 1995.

[7] Aue, A., Brey, M.:. Distributed Information Systems: an Advanced methodology, *IEEE TSE*, 20(8), pp. 596-605, 1994.

[8] GRADE V.2.0 (MS-Windows) GRAPES V3 (GRAPES-86+ GRAPES/4GL, GRAPES-BM). Sprachbeschreibung, Siemens Nixdorf, 1995.

[9] GRADE V.2.0 (MS-Windows). Modellierer. Benutzerhandbuch, Siemens Nixdorf, 1995.

[10] Barzdins, J., Kalnins, A., Podnieks, K. et al.: GRADE Windows: an Integrated CASE Tool for Information System Development, *Proceedings of SEKE'94*, pp. 54-61, 1994.

[11] Martin, J., McClure,C.: Structured Techniques: A Basis for CASE, Prentice-Hall, 1988.

[12] Rumbaugh, J.: Object-Oriented modeling and Design, Prentice-Hall, 1991.

[13] Barzdins, J., Barzdins, G. and Kalnins, A.:. Rules-Based Approach to Business Modeling, *Proceedings of SEKE'95*, pp. 164-165, 1995.

System and Business Process Re-engineering with GRADE

Jānis TENTERIS and Ēvalds VIĻUMS

Riga Information Technology Institute (RITI),
13, Skanstes St., Rīga, Latvia,
Internet: jtenteris@swh.lv, evilums@swh.lv

Abstract

This paper describes the main work packages (WP) performed during system and business process re-engineering with GRAPES (graphical specification) language and GRADE (Graphical Re-engineering Analysis and Design Environment) tool. The work packages are discussed mainly in their sequence within system life cycle. A short overview on diagram types for system modeling is included and the main features of GRADE tool are mentioned. The main concentration is on Business Process modeling and reengineering work package, which includes static analysis of business processes, simulation alternatives and principles of process and organization structure rearrangement. An example shows gradual improvement of business process model and its conversion to a person's job description and/ or Information System specification.

1 Sequence of Work Packages

System modeling and re-engineering with GRADE includes the following main work packages:

1. Registration and hierarchical arrangement of the main objects of the existing system;
2. Graphical representation of communication diagrams with main physical parts and functional objects of the system and communication between them;
3. Modeling of Business Processes:
   a) description of existing business processes, modification of existing processes and creation of new business processes:
      I. static analysis of business processes;
      II. simulation of business processes;
      III. evaluation and modification of processes according simulation results;
   b) modification of organization structure;
4. Development of data model, including:
   a) structured description of messages;
   b) contents of the databases;
   c) access rights to data;

GRADE-BM : Modeling and Simulation Facilities

A.Kalnins, J.Barzdins, A.Kalis

University of Latvia
Institute of Mathematics and Computer Science
Rainis Blvd. 29, Riga LV-1459, Latvia

and

RITI
Skanstes Str. 13, Riga LV-1013, Latvia

Abstract

The paper briefly outlines the business modeling language GRAPES-BM and the CASE tool GRADE-BM based on it. The business modeling language GRAPES-BM is a semiformal graphic language for modeling and simulation of complicated business systems (production processes, offices, information systems). GRAPES-BM relies on such basic concepts as task, event, performer, triggering condition etc. and contains advanced facilities for describing system behaviour. The main emphasis in the paper is on simulation facilities supported by GRADE-BM.

1. Introduction

In order to understand the behaviour of a system, it is necessary to understand activities within this system, causal links between these activities, their stimulus and results. In most cases the behaviour is described by diagrams consisting of symbols (rectangles, bubbles etc.) representing activities, various connecting lines representing the links and, possibly, some auxiliary symbols.

The first such formalism was Data Flow Diagrams [1], which were introduced for other purposes and only sometimes are used for BM. A number of similar, more or less specific formalisms followed; among them function dependency diagrams [2], event schemas [3], EPC diagrams [4. 5], Business process diagrams [6] etc. At the given moment none of the formalisms are universally accepted.

GRAPES-BM, described in this paper, is also a semiformal approach for describing behaviour and structure, using similar graphic notation as its basis. The main differences from all the abovementioned languages are in the following

- GRAPES-BM is oriented to semiformal description of arbitrary **Business Systems,** e.g. production systems, offices, enterprises, etc., not only **Information Systems**
- the level of formality: in GRAPES-BM it may vary from very informal use (as in traditional approaches [2,3,4]) up to a very formal nearly program like description of business systems.

The other important criterion is the possible tool support in the analysis of the described business system. Again, the more formalized is the approach, the richer set of tools is available. In most cases, e.g. [5], the tool support reduces to simple consistency checking, reporting and some static evaluation, most frequently, finding the critical path in a weighted flow graph. The other tools offer dynamic prototyping

and simulation as the main analysis method. Among the known BM systems Designer 2000 [6] should be mentioned.

GRAPES-BM with its support tool GRADE-BM is mainly dynamic execution oriented. Even very informal GRAPES-BM models may be executed in a sense, thus giving much deeper insight into system behaviour and its possible bottlenecks. On the other hand, for highly formalized models precise simulation of quantitative aspects is possible in a way close to specific simulation languages.

Development of GRAPES-BM started with A.Aue and M.Breu paper [7]. Afterwards M.Breu, A.Mraz, N.Richter, at al (European methodology and System Centre) have issued several preprints where the concept was developed further. The current version of GRAPES-BM and corresponding CASE tool GRADE-BM, described in this paper, is the result of collective work of many people. In the development of GRAPES-BM, besides the authors of this paper, the following people have made significant contribution (in alphabetic sequence): I.Etmane, D. Foerster, E. Knoener , K.Podnieks, C. Rositani, U. Sukovskis, A. Teilans, M. Weiss, A.Zarins, U.O. Ziemelis.

2. Main concepts of GRAPES-BM

Business modeling in GRAPES is based on two fundamental concepts: tasks and events.

2.1 Tasks

Any activity which is performed in a business system to be described is considered to be a task. Tasks may be very large - defining one basic activity of an enterprise and very small - like signing of a document. Large tasks are decomposed into chains of smaller ones using **Task Communication Diagrams (TCD)**. These diagrams are the basic ones in GRAPES-BM. Each task has its **name**. But it may have other formal attributes like

- performer of a task
- triggering condition
- duration.

Graphically (in TCD diagrams) tasks are represented by rounded rectangles. This rectangle shows the task name and its basic attributes, e.g. performer.

There are two types of tasks

- ordinary (transformation) tasks
- decision (branching) tasks.

Decision tasks have two or more named decision symbols attached to them. During execution one of the alternative outputs are chosen.
Examples of tasks are shown in Fig. 1

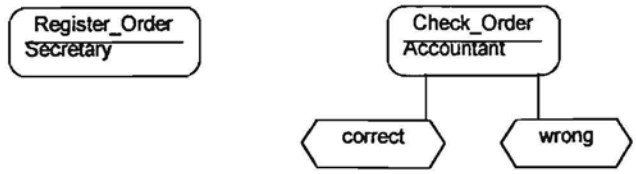


Fig. 1 Examples of tasks

It should be mentioned that the concept of task is present in any business modeling approach, only the terminology is quite different. Tasks are called functions in function hierarchy and dependency diagrams [2], process steps in ORACLE process diagrams [6], operations in Martin's OOA event schemas [3] etc.

2.2 Events

The other fundamental concept of GRAPES-BM is event. Events represent anything that can happen in a business system. Events are also the other principal element of Task Communication diagrams. They are represented by arrows leading from one task to another.

There are several categories of events:

- **message events**

Events with the category "message" correspond to objects produced by one task and transmitted to another. This concerns materials (e.g. paper, part of machine) and pure information (invoice, bill, report).
Message events always have **name** which is depicted next to the arrow.
Message events can carry information with them. The information is represented as a datatype associated with the event. The association is described in **Event Table (ET)** - an object global for the whole business model.

- **control flows**

They express the fact that one task is completed and the next task may start. Control flows are represented as unnamed arrows.

- **timer events**

These are the only events not created by tasks. They appear in certain time moments from an abstract timer which is represented as a small clock. Tasks can only receive timer events, in addition to message events from other tasks. Each timer event has a name. The exact definition of time moments for a timer is given in the Event table.
Fig. 2 shows examples of events as they are depicted in TCD diagram.

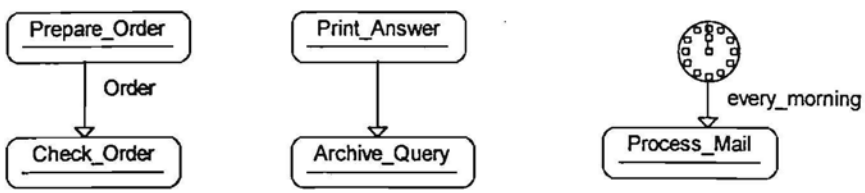


Fig. 2. Examples of events

One task can produce more than one event at the output. Similarly, one task can have several input events arriving from different tasks.

2.3 Task details

The next fundamental concept of GRAPES-BM language is **triggering condition**. It is associated with a task, as one of its essential properties. Triggering condition specifies which combinations of input events are necessary to start the task. This condition is specified as a boolean formula containing ANDs and ORs on event names e.g. *Order AND Payment.*

The general event semantics principle in GRAPES-BM is that incoming events form FIFO queues in front of a task (a separate queue for each event name). When triggering condition becomes true, the task consumes the relevant set of events from its input queues and starts execution. In the simplest case the triggering condition may be reduced to simple AND (or even "&" sign), which means ANDing all possible input events (i.e. one from each queue). Control flows also are implicitly ANDed in this case (i.e. all of them must be present). Similarly, simple OR (or "|" sign) means that anyone of input events (including control flows) is sufficient for triggering. If any of the required events is not present, the task waits for its arrival.

The next important part of a task is its **performer**. Performer specification consists of one or more performer names connected by AND and OR connectors. Performers may be organizational units, persons (positions) and equipment (resources). The available performers and their number are specified in the ORG diagram of the business model. The requested performers must be free before the task can really start, therefore the triggering condition is only the necessary condition for a task to start.

Duration of the task specifies the required execution time, e.g. task *Order_processing* in Fig. 3 takes exactly 1 hour.

Fig. 3 shows an example of completely specified task in a TCD.

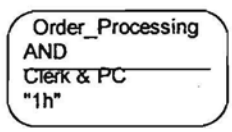


Fig. 3 Completely specified task

Once the task has started, it performs its main activity, which is not formally specified in GRAPES-BM. When the task is completed, it possibly takes one of its decisions and sends its output events.

We can describe more formal and informal details of a task in its **Task Specification Diagram (TSD)**. In particular, extended informal description of a task may be given there.

In conclusion one more remark on tasks. In any behaviour description there are tasks which are not part of the business system under consideration, e.g. a customer preparing an order. Such tasks are called **external tasks** in GRAPES-BM. They are represented using dashed lines for the task symbol.

Data manipulation in GRAPES-BM is described only at informal level. There are two special symbols for that purpose: **data store** for representing persistent data and **data object** for common data with life time of one transaction. No formal semantics is assigned to them.

3. Example of TCD diagram

The main goal of business modeling is to describe both readably and concisely a business system behaviour. As it was already pointed out, the main sort of diagrams for this purpose is Task Communication Diagram (TCD), describing a behaviour of large task in terms of smaller tasks and events.

Let us consider an example of a simple business system, namely, a simplified production line for producing printed boards. Fig. 4 presents this system as one TCD diagram. When necessary, the supervisor orders a new board. The operator takes an empty board and puts it into a robot, which has to assemble 10 parts on the board. Any of the parts may be faulty. The robot positions the board, takes one part from the appropriate parts store and assembles it on the board. All this is repeated 10 times. After that the completed board is tested by the tester (i.e. whether all parts are normal). If at least one part is found faulty, the operator tries to repair the board manually.

However, as in any production system, faults may be present anywhere. The testing is unreliable, the probabilities of correct test outcome are different for normal and faulty boards. The mean testing times also differ. To describe this situation more accurately, the testing task must be duplicated - one for a normal board and one for the faulty one. Symbols shown via bold lines are the ones having real counterparts in the production process. The other ones are so-called **technical** tasks introduced to describe the probabilistic nature of the production more accurately. These technical elements are insignificant to the modeller, they are vital only for simulation.

The diagram contains also some "data processing" elements - SET options for outgoing events, expressions on incoming event data in decisions, etc. From the modeller's point of view they may be read as comments. But they have precise semantics from the simulators point of view. This semantics will be explained in Section 5.

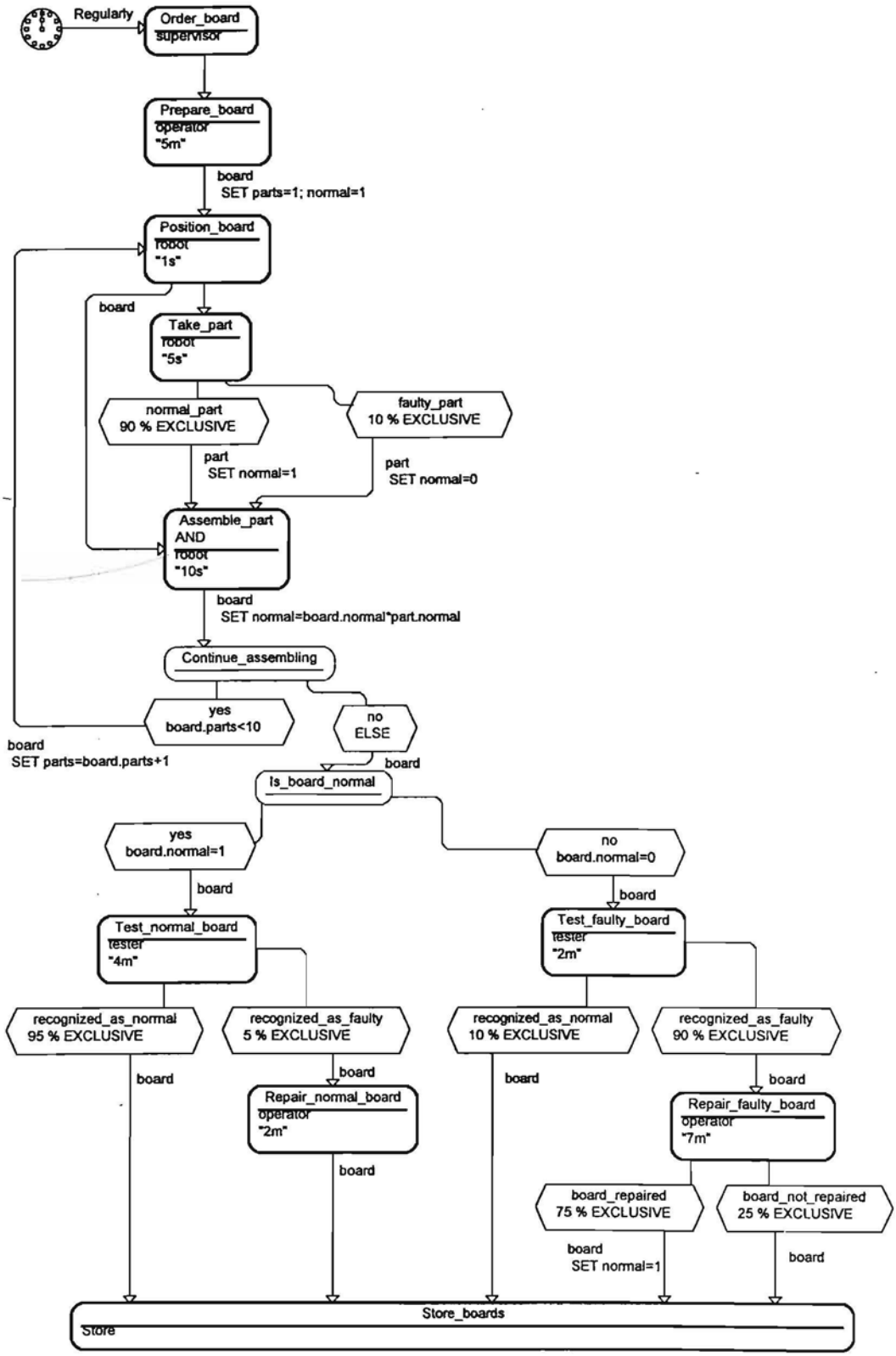


Fig.4 Example of TCD diagram

4. General structure of business model

The previous sections have described the most significant type of diagrams in GRAPES-BM - TCD diagram. Some other types of diagrams have been mentioned (ET, TSD). But there are more components in a business model.

4.1 Model tree

A complete business model is a hierarchy of the abovementioned (and some other) diagrams. The hierarchy itself is defined in a **model tree.** Fig. 5 shows the model tree for the production example. Model tree may be considered as a table of contents for the model.
The top line contains the elements global for the whole business model (ORG diagram, ET table, CMP table, SP table).

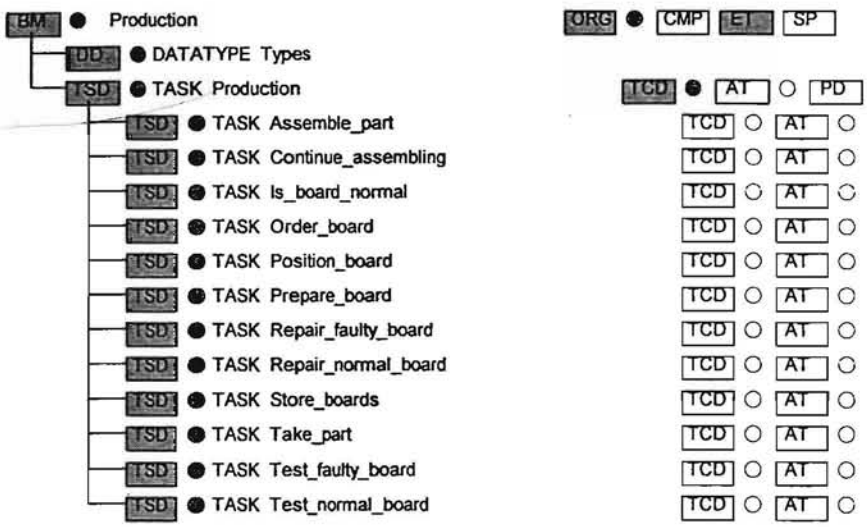


Fig. 5 Model tree

The main part of model tree is constituted by **primary tasks** (top-level TSD diagrams) and their refinements. There is only one primary task *Production* in the example. In general, primary tasks correspond to the main relatively independent functions of an enterprise. Each primary task normally is refined by its TCD diagram (business process), shown to the right of the corresponding TSD diagram.
The next level of refinement is defined by the set of subordinated TSD diagrams corresponding to all tasks mentioned in the TCD diagram.
The models with one primary task and one TCD diagram refining it are called **flat** models. However, in general case the situation is much more complicated. Each task in the TCD diagram may be further refined by its own TCD diagram (placed in the tree in the same line as the corresponding TSD diagram). The refinement is continued until we obtain the lowest level tasks which are called **elementary tasks**. For elementary tasks their formal and informal characteristics can only be specified in their TSDs. The choice of elementary tasks depends on the specific application of business modeling. The abovementioned language facilities show how traditional **structural refinement** is supported in GRAPES-BM.

4.2 TSD diagram, referenced tasks

There is a TSD diagram for each task appearing in a TCD diagram.
In the simplest case a TSD diagram contains the same formal attributes of the task as those visible in the TCD plus extended informal description of the task. In addition, the task's interface to its environment is also visible in TSD via so called **referenced tasks**. Fig. 6 shows the TSD diagram for the task *Take_part*. However, in general case, TSD diagram may contain significantly more information (briefly sketched in the next section) which is of special value for simulatable models.

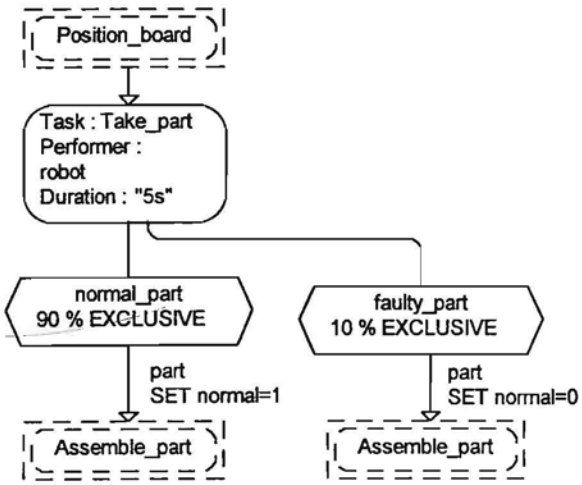


Fig. 6 Example of TSD diagram

Referenced tasks (with the relevant event arrows attached) in TSD correspond to task's neighbors in the corresponding TCD (whose part the task is). If the task is not elementary itself, the same referenced task symbols (containing the same names) reappear in the refining TCD diagram of this task (i.e. in the TCD one level below). Event arrows lead from these referenced task symbols to tasks in the refinement (or vice versa). The referenced task symbol is the key mechanism in GRAPES/BM for ensuring unambiguous event routing between the adjacent TCD levels in refinement. There are strict rules on refinement consistency (in most cases these rules are ensured by GRADE editors automatically).

4.3. Description of Organization Structure

The static structure of a business system is described by ORG diagram in GRAPES-BM. Fig. 7 shows the ORG diagram for the board production..

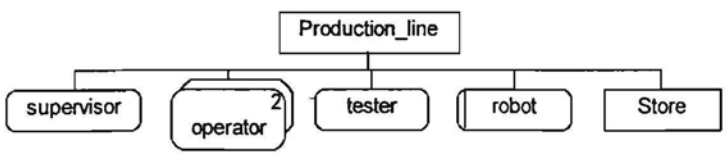


Fig. 7 Example of ORG diagram

Formally, ORG diagram may contain

- **organizational units** (company, department, group etc.)
- **positions** (chief, accountant, secretary, programmer etc.)
- **resources** (any kind of equipment, like car, PC etc.)

Any of the elements may be single or multiple, for multiple elements the number of available instances may be specified (otherwise unlimited number is available).

Organization structure is depicted as a tree (more precisely, as a set of trees) built from the abovementioned nodes. The edges of the tree represent **contains** relationship.

A leaf of a tree may be refined further by another tree.

Any of the organizational structure elements may have the following additional attributes:

- type (internal or external)
- competence list
- availability (as time interval)
- cost per hour
- efficiency level
- employee name (for a single position only)

All ORG diagram elements have also precise formal semantics, which is taken into account when TCD diagrams with performer specifications in tasks are executed.

4.4 Other tables and diagrams

Now some words on other elements of business model.

Event table (ET) has a row for definition of each event used in the business model. The most complicated are the timer definitions. GRAPES-BM provides formal means for timer definitions, e.g.

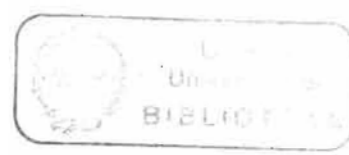
AT_5_PM could be defined as *TIME("*.*.* 17:00")*,
Regularly as *REPETITION("10m")*.

Message events may have their data types specified (record or elementary types may be used). Event types are necessary when data carried by events are taken into account. For example, the event *board* used in Fig. 4 has a record type with two integer fields *parts* and *normal*.

Competence table (CMP) is a supplement to ORG diagram, listing possible competences of ORG elements. **Simulation parameters (SP)** table contains general numeric parameters of the model.

One more type of diagrams to be mentioned is **attribute tables (ATR)**. They are global for the whole model, and there may be several named ATR tables. Each ATR table describes user-defined attributes for the given task type, which is equal to the ATR name.

Zero or more **Data Definition (DD)** diagrams are also global for the whole model. They describe data types used for events and are placed in the model tree above TSD diagrams.



5. Advanced Features of TCD and TSD diagrams

The items discussed so far have been more or less related to semiformal business system description and analysis.
However, GRAPES-BM permits to describe precise behaviour of business systems from the control point of view, including some data-related dependencies (as it actually is done in the *Production* example). This layer of GRAPES-BM actually constitutes a sort of process simulation language.
The basis of all these features is the assumption that events can carry data with them, and the data may be "processed" by tasks, used in decisions and transferred further to output events.
The following features are available:

- advanced triggering conditions, like
  - *Letter AND ALL Answer WHERE Letter.Id= Answer.Id*
  - *Letter AND <5> Comment*
- detailed description of task decisions
  - by their probabilities, in exclusive or nonexclusive manner
  - by precise formulas which may depend on data carried by the triggering input events and on numeric attributes of the task. For example, *board.parts* is the reference to the record field *parts* (containing the number of parts already assembled) in the event *board.* Built-in boolean function *Is_triggered_by (event)* may also be used in decisions.
- task duration dependent on its input data
- formulas for setting values of user-defined task attributes (then the task must have one of the types defined by ATR tables of the model). Formulas may reference data of input events and other attributes
- **SET**-option with formulas for setting data (record fields, as a rule) in output events of the task, with formulas referencing similar class of values. For example, *SET parts=board.parts+1* advances the event field *parts* (the "loop counter") by one. There is a special convention of data passing , namely, if an output event has the same name as an input event, data are passed without any formulas specified
- repetition factor for output events
- priorities in seizing performers.

These described features may appear in TCD as well as in TSD diagrams. The "computational aspects " are taken into account only for elementary tasks.

6. Semantics of Business Model, Concept of Transaction

6.1 Simple case with one TCD diagram

The example in Section 3 could be understood and even analyzed quite intuitively.
However, GRAPES-BM has a completely precise semantics defined. This semantics may be used for unambiguous manual validation of business models and for their execution by GRADE tools, i.e. simulation, prototyping, animation. It should be emphasized that Fig. 4 constitutes a syntactically correct and executable TCD diagram (certainly, in the context of the whole business model).

Timers in a TCD diagram are spontaneously active elements, i.e. they send their events to the appropriate tasks. These tasks are then triggered and afterwards they send their resulting events to other tasks. Thus the whole business process gets into motion. There may be as many concurrent instances of any task active as available performers permit it.
But there is one completely novel element added to this relatively straightforward semantics. This is the concept of **Transaction**.
Intuitively, a business transaction is a chain of activities initiated by some external stimulus and ended at the moment when further events are beyond our scope of interest. In the example of Fig. 4 the transaction starts with the arrival of a new order for a board. To be more formal, it starts from the moment when the timer *Regularly* starts the task *Order_board*. The transaction is completed when *Board* is sent to the task *Store_boards*.
The concept of transaction is fundamental in business modeling, since it helps to find out and analyze essential groups of activities inside a business system.
The main problem here is to find a simple formal definition of transaction which would coincide with the intuitive understanding in most cases.

In GRAPES-BM the following definition is used:

> The **transaction starts** solely when a task is started **only** by events coming from outside the business system.

Two types of events in GRAPES-BM are defined as such "outsiders":

- timers, described already in Section 4.4
- spontaneous events. Any event may be made spontaneous in the TSD diagram by assigning generator definition to it. These generator definitions have the same syntax as timer definitions. Spontaneous events are used to hide away a timer in a TCD diagram and to make the impression that the event comes right from an external task.

Thus, in Fig. 4, the timer *Regularly* starts a transaction since the task *Order_board* is triggered solely by it. But if there were another timer attached, e.g. to the task *Test_normal_board*, it would start no transaction, since this task requires also (internal) event *board* in order to be triggered.

The precise description of transaction behaviour is based on so called **Transaction Identifier (TID).** At the beginning of each transaction the starting event is given a unique TID. This number will be used throughout the transaction, all events and tasks in the corresponding task chain will be tagged by it.
There is no explicit use of TID. However, it participates implicitly in each triggering condition. AND condition will be true only if all incoming events have the same TID. Thus only matching groups of events that belong to the same transaction can trigger a task. In Fig. 4 only that *part* which corresponds to the *board* for whose assembling it was taken, will trigger the task *Assemble_part*.
Transaction is completed when there are no more events in the model with the given TID. In some occasions default rules are insufficient. To cope with these situations the following options of tasks may be used:

- NOSTART for preventing an unwanted start of transaction
- START for explicit start of transaction
- END for explicit end
- NOTID for explicit stripping off the TID from an event.

6.2 Semantics in general case

When there are several levels of TCDs in the model, the key "players" are elementary tasks. For formal semantics definition, it should be assumed that all non-elementary tasks are expanded via their refinement TCDs. Thus a virtual flat TCD would be obtained. The correct event routing in this TCD is defined according to strict rules based on referenced task symbols. The general semantics of model behaviour is the "simple" semantics for this flat TCD.

However, structuring adds a new dimension to transactions also. In general, there may be a transaction level for each non-elementary TSD (or for each TCD level, which means the same). There are precise default rules for simultaneous start of nested transactions of all possible levels.

In addition, for the lowest level transaction the default start rules have become more complicated:

- besides timers and spontaneous events, any event coming from a referenced task symbol is considered as an outsider
- when an event leaves the relevant TCD diagram (which corresponds to transaction under consideration) via an outgoing referenced task, it is stripped off its TID.

The default start rules for simultaneous start of several levels are defined so that strict nesting of transactions is always preserved. In short, the default start of a level occurs, when an event just passes through the level via referenced task symbol.

For each level of a transaction there is a separate TID, thus events actually are tagged by lists of TIDs. All these TIDs are taken into account, level by level when an AND condition on a set of events is checked. For TIDs of each level there are natural boundaries where they are stripped off. The default end condition for a transaction is the same as in one-level case.

For explicit transaction control, START and END options may contain lists of transaction (task) names.

7. Expressibility of GRAPES-BM

7.1 Theoretical aspects

From the theoretical point of view, GRAPES-BM language is close to some well-known extensions of Petri nets, for example to so-called coloured Petri nets (CPN) (see, e.g. [10], [11]). Tasks correspond to transitions in CPN, event queues (invisible in GRAPES diagrams) to places, triggering conditions to guards and arc expressions etc. Many natural examples look quite similar in both formalisms. Structuring is also similar in them. The main difference is that in GRAPES-BM there are implicit event queues at each elementary task, while in CPN places are explicit pools (not queues) common to several transitions. GRAPES-BM has a more advanced timing control. Any formal comparison of these formalisms is extremely difficult due to complexity of both of them. CPN formalism was designed with reachability and other kinds of static analysis in mind (e.g. for finding deadlocks). The formal semantics of

GRAPES-BM was defined with the goal of easy behaviour simulation of complicated business system.

The comparison shows that formal semantics of GRAPES-BM could be defined as rigorously as it is done for Petri nets. Up to the moment the semantics of GRAPES-BM is defined informally, but precisely enough to build a simulator (interpreter) for it and to make it really usable language for building large executable models. Certainly, the precise definition is beyond the scope of this paper.

7.2 Practical aspects

From the practical point of view GRAPES-BM allows as to describe adequately various control structures present in business systems, like:

- iterative looping depending on event data
- centralized control depending on some global data
- time-out control of incoming events

The "programming " of such control structures is sufficiently simple. The methods used remind slightly those used in "programming " of Petri nets. Here is significant role played also by transaction concept. In most cases the precise control aspects may be simply added to models originally built for pure qualitative analysis. A simple case of loop programming is demonstrated in the example of Fig. 4 with *board.parts* being the loop counter. The "programming" of such control structures uses so-called "technical" tasks and events - ones which have no counterparts in the real world system to be described.

The "programming" of any centralized control affecting several tasks and/or their instances is based on the following general construct (see Fig. 8).

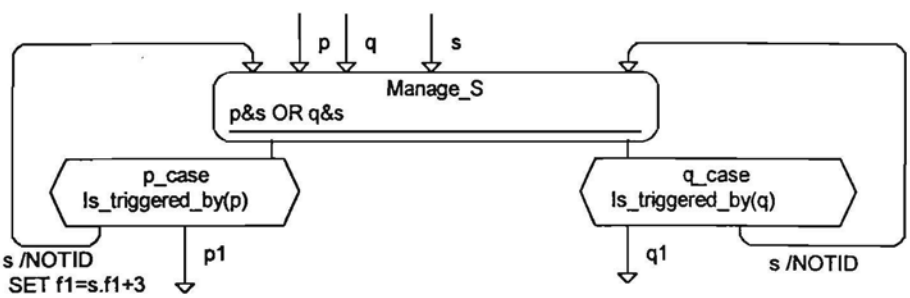


Fig. 8 General control structure

There is an event *s* representing the current status of some control object. The object is "managed" by the task *Manage_S*. The numeric components of the object, e.g. the currently available amount of some resource *r1*, are represented by fields (e.g. *f1*) within the record datatype of the event *s*. Events *p* and *q* represent requests to the control object from tasks, and *p1*, *q1* are the corresponding responses. The selectivity of a response is guaranteed by *is_triggered_by* built-in boolean function used in decisions. If there is an additional control logic dependent on resource values necessary, decisions may be extended by other formulas, e.g. *is_triggered_by (P) AND s.f1>10*. Certainly, then there should be more than two decision branches at the task *Manage_S*. The updated value of the status event *s* is returned to the same task after all decisions. NOTID option guarantees that this centralized control can coexist with requests from many transaction instances (but the relevant TID is retained in the response). The presented general schema may be easily adapted to various types of

centralized controls. In practical GRAPES-BM "programming" the transaction concept also plays a significant role, e.g. in building time-out control constructs.

8. BM-Simulator and other GRADE Modeller Tools

The full support of GRAPES-BM language is included in the new version of GRADE-BM Modeller toolset.

The following components of GRADE are available:

- tree and repository management
- advanced graphical editors
- syntax analyzer
- GRAPES-BM language interpreter (BM-simulator)
- animator
- trace browser.

In this paper we concentrate on language execution aspects of GRADE. We mention only that graphic editors play a key role in GRAPES-BM usage, ensuring real "graphic programming" with the features of

- smart automatic layout generation
- relevant name and syntax prompting
- automatic transfer of information from diagram to diagram.

Note. All diagram examples in the paper, including Fig. 4, have been produced by GRADE-BM editors.

Prior the execution, diagrams are processed by syntax analyzer, which besides syntax checking generates the intermediate code (in parse-tree format) for all relevant diagrams.

The heart of language execution is the BM-simulator.. It has the following features:

- step mode with variable granularity for business model dynamic debugging and step-by-step exploration
- run mode for business model prototyping and simulation. The run mode is combined with pause and breakpoint features
- inspect facility for observing any elements of the current status of business model (active tasks, event queues, data contained in them, etc.)
- user-controlled automatic statistics gathering (for predefined statistical features of tasks, events and performers and for statistics of user-defined task attributes)
- interface to diagram animator

Since even quite informal BM models are executable as a rule, the BM simulator serves as a powerful tool for model validation, step mode execution allows to find any unexpected behaviour of the model. On the other hand, a normal animated run of a model is very helpful in general evaluation of the model and in finding unexpectedly long queues and other bottlenecks in the system.

The BM simulator is based on the model activity calendar. The ground activities are

- insert a new event in a queue
- check the value of a triggering condition
- start a new task instance
- end a task instance

The main non-trivial problem is to implement a fair and effective performer allocation strategy, compliant with the described GRAPES-BM semantics. If the requested performers are busy at the moment when triggering condition becomes true, the potentially triggerable task is enqueued at these performers which would satisfy the performer expression. When a performer is released, tasks enqueued at it are checked whether they can really start. It should be noted that this strategy gives some preference to tasks with smaller performers sets. Certainly, the explicit PRIORITY clause in TSD permits the user to control the scheduling.

In general, the simulator solutions have guaranteed acceptable performance in both speed and model size. In typical Windows environment, on a 486-based machine, models with thousands of tasks may be processed, with several hundred thousands of simultaneously active task instances.

9. Conclusions

The language GRAPES-BM version 2.0 and the corresponding tool GRADE-BM version 2.0 were released in 1995 [8,9]. GRAPES-BM and GRADE-BM, described in this paper, correspond to the version 3.0. The differences from the previous language version are significant.

The version 3.0 of GRADE-BM in this moment is under development.

The first experimental applications of GRADE version 3.0 (design process management in car industry, some banking applications, public utility management et al) have shown the correctness of all principal decisions in tool set design. The size of business models which may be processed in typical Windows environment seems to be quite acceptable in practice.

The main novel feature of GRAPES-BM seems to be the wide spectrum of applicability of the same models, from general informal evaluation of the current system to numeric experiments with it

References

[1] T.DeMarco. *Structured Analysis and System Specification*, Prentice-Hall, 1979

[2] R.Barker, C.Longman. *CASE*METHOD Function and Process Modeling*, Addison-Wesley, 1992

[3] J.Martin, J.Odell. *Object-Oriented Analysis & Design*, Prentice-Hall, 1992

[4] G.Keller, M.Nüttgens, A.W.Scheer. *Semantische Prozessmodellierung auf der Basis Ereignisgesteuerter Prozessketten (EPK),* in Veroffentlichungen des Instituts fur Wirtschaftsinformatik, v. 89, Saarbrucken, 1992

[5] W.Brenner, G.Keller (Eds) *Business Reengineering mit Standartsoftware*, Campus Verlag, Frankfurt, 1995

[6] *Designer-2000. A Guide to Process Modeling.* Oracle Corp., 1995

[7] A.Aue, M.Brey. *Distributed Information Systems: an Advanced Methodology*, IEEE TSE, 20(8), pp. 596-605, 1994

[8] *GRADE V.2.0 (MS-Windows) GRAPES V3 (GRAPES-86+ GRAPES/4GL, GRAPES-BM). Sprachbeschreibung*, Siemens Nixdorf, 1995

[9] *GRADE V.2.0 (MS-Windows). Modellierer. Benutzerhandbuch*, Siemens Nixdorf, 1995

[10] K.Jensen. *Coloured Petri Nets*, in Advances in Petri Nets, 1986, LNCS v.254, Springer, 1987

[11] K.Jensen. *Coloured Petri Nets: High Level Language for System Design and Analysis*, in Advances in Petri Nets, 1990, LNCS v.483, Springer, 1991

АВТОМАТИЗИРОВАННОЕ ТЕСТИРОВАНИЕ ТЕЛЕКОММУНИКАЦИОННЫХ СИСТЕМ

М.Я. АЛБЕРТС, доктор математических наук, ведущий исследователь
А.А.КАЛНИНЬШ, доктор вычислительных наук , ведущий исследователь
Д.А.КАЛНЫНЯ, доктор математических наук, ведущий исследователь

Институт математики и информатики Латвийского университета
бульв.Райня 29, LV-1459, г.Рига, Латвия

Рассматривается попытка систематического тестирования программного обеспечения для систем мобильной радиотелефонной связи, используя RIGA-SDL и генератор тестов. Предлагается автоматический генератор тестов, базированный на изучение пространства глобальных состояний. В статье описаны методы уменьшения размера пространства глобальных состояний, которые позволяют решить проблему автоматической генерации тестов при ограниченных технических ресурсов. Кратко описана общая стратегия тестирования системы.

Ключевые слова: Генерация тестов, глобальное состояние, полное покрытие ветвей, утверждения, SDL.

1.ВВЕДЕНИЕ

Тестирование всегда бывает большой проблемой при разработке программного обеспечения, особенно для параллельных систем и систем реального времени. Статья представляет попытку решения этой проблемы в одном практическом случае, а именно, для систем мобильной радиотелефонной связи. Главным аспектом при этом является систематическое комплексное тестирование разрабатываемого программного обеспечения. Поскольку разработаны и в проекте использованы некоторые специализированные протоколы , то требуются также некоторого рода валидация и верификация (нахождение возможных тупиков и.т.д.). В качестве базисного программного обеспечения использована система поддержки SDL RIGA-SDL [1,2] на персональном компьютере типа IBM PC, так как мощные рабочие станции не были доступны.

За основу взят экспериментальный автоматический генератор тестов в системе RIGA-SDL, впервые представленный в [2]. В настоящей статье описаны значительные

улучшения этого генератора, сделавшие его практически применимым. В ситуациях, когда размер системы не позволяет применять полностью автоматические методы, большой интерес представляет методология контролируемого вручную процесса выбора тестов. В статье раскрыта методика, как ограничить поиск в выборе тестов до реалистических размеров, сохраняя полноту тестирования на приемлемом уровне.

Эта методология включает принципы построения соответствующих тестовых драйверов (в статье называемых тестерами). Описаны также некоторые эвристические методы уменьшения пространства глобальных состояний при переборе во время процесса выбора тестов. Для этого главное - наилучшим образом выбрать глобальное состояние исследуемой системы.

В последние годы достигнут значительный прогресс в развитии классических методов валидации, основанных на исследовании пространства глобальных состояний, особенно в протоколах [3,4,5,6]. Так как исследование пространства глобальных состояний лежит в основе и нашей автоматической генерации тестов, то в статье попутно описаны и некоторые элементы автоматической валидации . Они включают автоматическое нахождение тупиков (*deadlocks*) и неправильной маршрутизации сигналов, распознавание ошибок данных. Имеется специальный механизм для поддержки локальных утверждений, написанных проектировщиком. Можно использовать также аналог диаграмм последовательностей сообщений (Message Sequence Charts, в дальнейшем MSC) для автоматического нахождения аномалий последовательностей сигналов во время генерации тестов. При валидации главное внимание уделено на снятие добавочной нагрузки со столь критического исследования пространства глобальных состояний во время выбора тестов. Главное отличие между классической валидации [3,4] и нашим подходом в том, что мы пользуемся значительно меньшим пространством глобальных состояний.

Представляется также практический опыт генерации тестов для программного обеспечения системы и использования генерированных тестов в ее тестировании.

Статья содержит пять главных разделов. В разделе 2 описывается проектируемое программное обеспечение коммуникационной системы и процесс его тестирования. Раздел 3 дает теоретическую основу выбора тестов согласно [2] и новые разработанные ручные процедуры оптимизации исследования пространства глобальных состояний при

выборе тестов. Раздел 4 посвящен некоторым возможностям автоматической валидации, включая тех, которые связаны с MSC. Раздел 5 представляет короткое описание использования тестового генератора, а раздел 6 - первые полученные результаты.

2. ИССЛЕДУЕМАЯ СИСТЕМА И ЕЕ ТЕСТИРОВАНИЕ

2.1.Разработка программного обеспечения для системы мобильной телефонной связи

В последние три года группа сотрудников Института математики и информатики Латвийсского университета разрабатывает программное обеспечение системы мобильной радиотелефонной связи (*digital trunked radio*) по проекту с фирмой DTR International Inc., USA (США). Разрабатываемая система является типичной системой радиотелефонной связи с динамическим выделением радиоканалов [7].

Базовые станции поддерживают два режима работы

- аналоговый разговорный радиотракт с простым цифровым каналом сигнализации
- полностью цифровая коммутация со сложным протоколом канала сигнализации.

Возможен выход на публичную телефонную сеть. Базовая станция состоит из элементов, каждый из которых содержит блок приемника-передатчика и устройство управления на базе IBM совместимого персонального компьютера. Все элементы базовой станции соединены внутренней сетью передачи данных.

Все программное обеспечение для этого проекта (за исключением некоторых аппаратурных интерфейсов низкого уровня) разработана в SDL, используя систему RIGA-SDL. Использована версия языка SDL-88 с некоторыми расширениями, описанными в [1]. Некоторые из этих расширений, например, массивы блоков для представления групп идентичных управляющих устройств часто используются, и даже типы блоков и экземпляры в SDL-92 были бы менее пригодны для описания таких структур. В рамках настоящего проекта проведено некоторое улучшение системы RIGA-SDL, в том числе переход к Borland Pascal 7.0 в качестве базового языка и использование его защищенного режима (*protected mode*). Тестирование и отладка программного обеспечения происходят только на уровне SDL. Существенным улучшением в системе RIGA-SDL являются средства генерации кода для целевой среды, состоящие из модифицированного компилятора с SDL на Pascal и совершенно новой библиотеки поддержки выполнения для работы в реальном времени и в многопроцессорной среде.

2.2. Стратегия тестирования для проекта

Как в любом проекте программного обеспечения реального времени, тестирование программного обеспечения состоит из двух уровней. Нижний уровень, состоящий из тестирования технических интерфейсов и аспектов реального времени, в настоящей статье не рассматривается. Изучен верхний уровень - логическое тестирование программного обеспечения при общих ограничениях из-за параллельности и тестирование соответствующих протоколов. Сделано это исключительно на уровне SDL. При разработке программного обеспечения использованы два уровня описания тестируемой системы на SDL. Простейший из двух уровней SDL описания системы, называемый эталонной моделью (*reference model*), описывает управляющие алгоритмы и протокол, употребляемые в системе, включая их данные, но опуская детали связи с реальной техникой и данные, необходимые для поддержки этой связи. Эталонная модель может быть рассмотрена как спецификация протокола. Второе более сложное описание системы называется моделью реализации (*implementation model*). Эта модель содержит полный текст SDL для генерации кода для целевой среды. Обе модели содержат то же самое множество процессов, сигналов и даже те же самые состояния в процессах. Но некоторая обработка данных в эталонной модели упрощена, и некоторые вызовы процедур опущены. Обе модели имеют одинаковую структуру блоков (см. Рис.1).

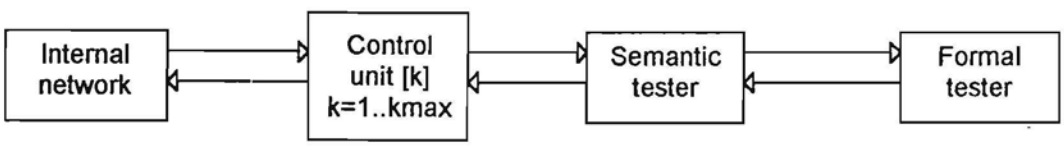


Рис.1. Структура системы

Управляющие устройства (*control unit*) в базовой станции представлены как массив из *kmax* одинаковых блоков. Каждый из блоков соединен с внутренней сетью (*internal network*). Остальные два элемента в Рис.1 представляют тестовые драйверы (*semantic tester* и *formal tester*).

Для автоматической генерации тестов используется эталонная модель. Полученные тесты могут быть применены к обеим моделям, так как тестовые драйверы являются общими для обеих моделей.

Главной целью в генерации тестов является полное покрытие ветвей (*complete branch coverage*) для некоторых существенных процессов в эталонной модели. Это означает, что для любой ветви типа *input, if* или *case* в выбранных диаграммах процессов существует последовательность внешних сигналов из множества тестов, выполнение которой влечет выполнение этой ветви .

Когда тест, автоматически генерированный для эталонной модели, выполняется вручную, полная последовательность входных/выходных сигналов может быть регистрирована вместе с самым тестом. Эта последовательность может быть использована как эталон вида MSC, применяя тот самый тест к модели реализации.

2.3. Тестовые драйверы

В обеих моделях имеются два тестовых драйвера - семантический и формальный. . Семантический тестер моделирует техническую внешнюю среду для управляющих устройств в очень простом виде, учитывая только существенные логические интерфейсы. Поскольку главным акцентом является логическое тестирование программного обеспечения управляющих устройств, то модули мобильных переговорных устройств (в реальной жизни связаны на уровне аппаратуры) представлены как простые заглушки (*stubs*) внутри блока семантического тестера . В этом блоке описано только логическое поведение мобильных устройств (в дальнейшем мобилей), учитываемое при тестировании. То же самое в силе относительно соединений с публичной телефонной сетью *(PSTN)*.

Формальный тестер обеспечивает только любой возможный логический стимул к системе (вызов мобиля из другого мобиля при свободном управляющем устройстве, вызов мобиля при занятом управляющем устройстве, вызов свободного мобиля из телефонной сети, окончание соединения и. т. д.). Сигналы с формального тестера не имеют параметров, семантический тестер автоматически обеспечивает нужные значения параметров (номер управляющего устройства, номер мобиля и.т.д.) в зависимости от требуемого контекста. Это значительно упрощает автоматическую генерацию сигнальных последовательностей с формального тестера.

Напомним, что нормально только закрытые SDL системы могут быть выполнены при помощи RIGA-SDL. Но на время генерации тестов в исследуемой системе один блок или процесс должен быть определен в качестве внешней среды. Он исключается из системы и его сигналы используются для перебора во время построения пространства глобальных состояний. В рассматриваемой системе в качестве внешней среды определен блок формального тестера.

Семантический тестер имеет также возможность контролировать параллельность обработки сигналов в системе. В простейшем режиме в каждый момент обрабатывается только одна транзакция, т.е., следующий стимул применяется к системе после получения всех моментальных реакций системы. В действительности в системе еще есть некоторая параллельность, потому что предыдущие стимулы могут быть причиной задержанных (т.е., зависящих от внутренних таймеров) действий. Второй режим обеспечивает полную параллельность, когда несколько стимулов обрабатываются одновременно. Возможен переход от одного режима к другому, меняя значения некоторых таймеров в семантическом тестере.

Еще одно применение семантического тестера - включение некоторой автоматической проверки сигнальных последовательностей для обеспечения валидации на базе MSC (см. 4.2.).

3. АВТОМАТИЧЕСКИЙ ВЫБОР ТЕСТОВ

3.1.Общие принципы

Теоретические основы для метода генерации тестов, использованного в этом проекте, описаны в [2]. Напомним коротко о них. Глобальное пространство состояний для некоторой системы SDL (как и в [3]) строится в форме графа достижимости (*reachability graph*), вершинами которого являются глобальные состояния, а ребрами - операторы диаграмм, выполнимые в глобальном состоянии, из которого ребра выходят. В каждой новой вершине этого графа проверяется, достигнута ли цель тестового покрытия (*testing coverage goal*). Цель покрытия - это достижение любой ветви в диаграммах указанных процессов/процедур [2]. Была ли достигнута какая нибудь новая ветвь, выполняя последовательность входных сигналов, ведущую к активной вершине графа достижимости, проверяется при повторении глобального состояния. Если да, то

соответствующая последовательность регистрируется как тест . Последовательность становится тестом также при нахождении некоторой аномалии.

Формально глобальное состояние системы SDL должно содержать состояния всех экземпляров процессов, значения переменных процессов, очереди внутренних сигналов и множество активных таймеров. Главной целью в [2] было уменьшение пространства глобальных состояний, в то же время гарантируя, что при этом любая достижимая ветвь остается достижимой. Для этого были введены понятия существенных операторов (СО) и существенных переменных (СП). Грубо говоря, в любом цикле в диаграмме выбирается один СО . Переменная является СП для данного СО, если ее значение в СО имеет прямое или косвенное влияние на некоторое решение типа *if* или *case* в диаграмме. В [2,8] дан формальный алгоритм для нахождения множества СП для данного СО по описанию процесса на SDL. В [2] показано, что, если глобальные состояния регистрируются только в СО и глобальное состояние содержит только существенные переменные (включая внутренние очереди сигналов и множество активных таймеров), то построение графа достижимости остается корректным - любая достижимая ветвь достигается в этом процессе.

Далее описаны разработанные на данной теоретической основе методы оптимизации генерации тестов, позволяющие успешно тестировать вышеописанную систему.

3.2. Ручная оптимизация глобального состояния

Для реальных систем, в том числе и для рассматриваемой системы мобильной связи, и для доступной технической платформы отношение множества практически достигаемых глобальных состояний к размеру всего графа достижимости обычно бывает слишком малым. В таких случаях, в терминах [3], управляемое частичное исследование множества глобальных состояний может быть значительно лучше неконтролированного частичного исследования (т.е., полученного при имеющихся ресурсах). Управление достигается при помощи дальнейшего эвристического уменьшения списков СП и СО. Это уменьшение проводится вручную, используя некоторые сведения о поведении тестируемой системы.

Возможны два случая:

- Первичной целью является некоторая автоматическая валидация системы, попутно получая автоматически генерированные тесты. В этом случае проверка дополнительных утверждений корректности является очень важным элементом.
- Первичной целью является автоматическая генерация тестов согласно вышеописанному критерию.

В первом случае построение графа достижимости должно оставаться корректным для исследуемой SDL системы, поэтому рекомендуются следующие, зависящие от этой системы оптимизации глобального состояния и списка СО:

* поскольку при обработке одного входного стимула в корректной программе SDL бесконечных циклов не должно быть, то во всей системе могут быть выбраны в качестве СО только некоторые состояния процессов. Такие СО в действительности становятся выбранными вручную точками прерывания (как при отладке системы);
* если в системе некоторые СП имеют равные значения или они функционально зависимы, то в качестве СП выбирается только одна из них;
* если содержания некоторых очередей внутренних сигналов функционально связаны со значениями некоторых переменных, то лучше в глобальном состоянии держать не очереди, а значения этих переменных;
* если состояние процесса однозначно определяет значение некоторой СП в этом процессе, то в глобальное состояние следует поместить либо состояние процесса, либо значение переменной;
* в глобальное состояние можно не поместить внутренние состояния вспомогательных процессов, не имеющих влияние на главные процессы системы.

Все эти законы только эвристические. Их строгое соблюдение требовало бы суждения о системе, похожего на доказательство корректности, поэтому они могут быть использованы только как семантические указания. Опыт авторов показывает, что этими законами можно пользоваться, имея минимальные знания о поведении системы, и что они имеют огромное влияние на уменьшение размера пространства глобальных состояний до реального для полного исследования системы на доступной технике *(hardware platform)*.

Во втором случае, когда генерация тестов является главной целью, глобальное состояние надо выбрать по возможности минимальным, лишь бы не терялась достижимость ветвей процесса. Хорошим началом тут может быть включение в глобальное состояние только состояний существенных процессов и исключение данных процесса. Если покрытие ветвей при этом окажется недостаточным, то к глобальному состоянию добавляются некоторые данные. Обычно это необходимо в случае, когда актуальные состояния протокола кодируются не соответствующими состояниями процесса, а некоторыми переменными. Именно эти переменные и должны быть включены в глобальное состояние. Опыт показывает, что этого достаточно для автоматического получения полного покрытия ветвей. В качестве СО рекомендуется выбрать такие операторы состояния в семантическом тестере, где происходит ввод внешних сигналов. Тогда ребрам в графе достижимости соответствуют внешние сигналы из формального тестера. Надо отметить, что проверка утверждений сохраняет смысл и в этом случае.

3.3. Организация перебора и выбор тестов

Когда глобальное состояние определено, то для строения графа достижимости можно использовать любой классический метод исследования пространства глобальных состояний [3]. В нашем случае наилучшим подходом является поиск "сперва в глубину" (*depth-first search*) с полным (*complete*) сохранением созданного пространства глобальных состояний..

Поиск "сперва в глубину" ставит вопрос о выборе максимальной глубины для каждой исследуемой системы. При этом можно использовать только эвристические утверждения, например, для тестирования группы из n управляющих устройств достаточно тестовых последовательностей длины n+1.

В процессе строения графа достижимости определяются тесты по вышеописанному правилу. Достижение новых ветвей диаграммы проверяется как при совпадении глобальных состояний, так и при максимуме глубины поиска. Этот подход дает приемлемое количество тестов приемлемой длины.

4.ВАЛИДАЦИЯ ВО ВРЕМЯ ГЕНЕРАЦИИ ТЕСТОВ

4.1. Использование утверждений

Утверждения, относящиеся к некоторым точкам в диаграмме, определяются просто. Точка должна быть выбрана в качестве СО (точки прерывания) и каждый раз, когда во время исследования множества глобальных состояний эта точка достигается, автоматически проверяется утверждение. Утверждение должно быть написано как паскалевская булевская функция с возвратом истины в нормальном случае. Каждый случай, когда утверждение оказалось ложным, автоматически регистрируется и во время исследования множества глобальных состояний соответствующая последовательность внешних сигналов сохраняется. Существенным использованием утверждений является дополнительная проверка аномалий поведения. Например, частичные тупики, касающиеся только некоторых процессов, характеризуются тем, что их очереди входных сигналов пустые и другие процессы находятся в состояниях, в которых в данной ситуации не могут быть посланы сигналы в тупиковую часть. Утверждения сознательно надо отделить от SDL-описания самой системы, чтобы не мешать генерации эффективного кода для целевой среды.

4.2. Моделирование валидации на базе MSC

RIGA-SDL не имеет явной поддержки MSC как средства спецификации. Для MSC нет редактора, но тестовые результаты могут быть показаны в форме MSC. Поэтому только неявная поддержка для валидации на базе MSC может быть обеспечена.

Ограниченная валидация на базе MSC реализирована на двух уровнях. Во первых, некоторые действия надо включить в семантический тестер. В простейшем случае семантический тестер в каждый момент времени требует выполнения только одной транзакции (т.е., стимул - группа моментальных системных ответов). Тогда фрагмент MSC, определяющий, какие ответы должны быть получены на данный стимул, может быть проверен семантическим тестером (действительно получены ли все моментальные ответы на данный стимул). В описанном применении ответные сигналы должны фильтрироваться согласно их параметрам (т.е. номером управляющего устройства), так как бывают запоздалые ответы прежних стимулов.

Во-вторых, результаты проверки должны быть помещены в переменной, которая используется в некотором утверждении. Таким образом, автоматический отчет о MSC аномалиях вместе с соответствующими последовательностями входных сигналов создается во время генерации тестов. Время поступления ответов может быть проверено подобным образом.

5. РЕАЛИЗАЦИЯ СРЕДСТВ ТЕСТИРОВАНИЯ

Закончена первая версия практически используемого генератора тестов на базе RIGA-SDL, за основу которого взят экспериментальный генератор тестов, описанный в [2] и имеющий несколько значительных улучшений. Одним из факторов практической используемости является новая быстрая операция сохранения/возобновления полного состояния системы SDL при выполнении. В определении пространства глобальных состояний имеются больше возможностей. Оптимизировано хранение активных вершин графа достижимости.

Далее описаны некоторые детали работы с тестовым генератором. Во первых, строится описание системы на SDL и происходит ее ручная отладка для устранения грубых ошибок. Следующий шаг - подготовка заказа генерации тестов (*test generation request*). Указывается, которые процессы должны быть включены в критерий покрытия ветвей, а также процессы, состояния которых следует включить в глобальное состояние. Задается точки прерывания, максимальная глубина поиска и некоторые другие параметры. Указывается блок, выбираемый в качестве внешней среды (в нашем случае это формальный тестер). Разработчик еще должен написать специальные паскалевские булевские функции (*user functions*) для каждой точки прерывания, представляющие утверждения в этих точках. При этом можно использовать как переменные процессов, так и специальные средства доступа к состояниям процессов и очередям сигналов. Кроме этого, значения тех переменных процессов, которые разработчик выбрал для включения в глобальное состояние, следует поместить в специальные фиксированные переменные для использования при строении текущего глобального состояния.

Как видно из Рис.2, сначала SDL-описание тестируемой системы обычным образом обрабатывается анализатором и генератором кода в системе RIGA-SDL. Кроме паскалевского кода исследуемой системы паскалевский код создается также в результате применения процессора заказа (*request processor*). После этого для получения выполнимой задачи тестирования (*executable test generator)* происходит компиляция и сборка *(linking)* вместе с подпрограммами поддержки SDL и программой генерации тестов *(test generation kernel)*. Выполнение этой задачи *(run of test generator)* дает как тесты *(tests*), так и сообщения об аномалиях *(anomaly reports)*. Во время генерации тестов создается необъязательный протокол генерации, позволяющий потом анализировать поведение тестируемой системы.

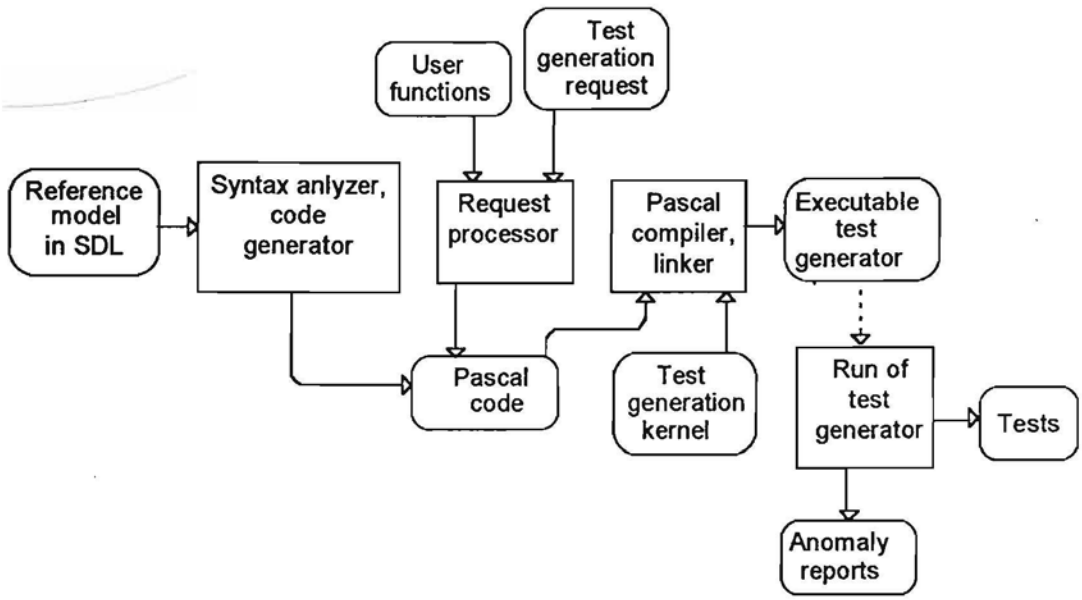


Рис.2. Схема генерации тестов

6.ПЕРВЫЙ ОПЫТ ГЕНЕРАЦИИ ТЕСТОВ

Первая успешная попытка генерации тестов проведена для упрощенной системы мобильной радиотелефонной связи. Эталонная модель системы содержит 18 процессов с общим числом состояний 37. Требуется полное покрытие ветвей для двух процессов системы с общим числом ветвей 96. Написан соответстующий семантический тестер, выполняющий свою базовую функцию - нахождение параметров входных сигналов, передаваемых системе мобильной связи. Выбрана одна точка прерывания в семантическом тестере. Рассматриваются только соединения от мобиля к мобилю *(mobile-to-mobile communications)* и это осуществлено 5 различными входными сигналами из

формального тестера. Глобальное состояние определено 3 различными способами. Первым выбрано самое естественное определение - регистрируется, в каком состоянии находится главный управляющий процесс во всех управляющих устройствах. Во втором случае регистрируется только массив переменных семантического тестера, определяющий состояния всех управляющих устройств, опуская состояния процессов. В третьем случае регистрируется суммарная информация вышеупомянутого массива - число управляющих устройств данного типа, находящихся в данном состоянии .

Для всех трех определений глобального состояния были достигнуты все выполнимые ветви в выбранных существенных процессах, и получено одинаковое число тестов. Как видно из таблицы 1, третье определение уменьшает пространство глобальных состояний. Время генерации тестов на IBM PC с процессором типа 486 средней скорости было от 2 до 4 минут .

В результате тестирования были найдены 2 ошибки, каждая из которых создала тупиковую ситуацию в редко использованном случае.

На втором этапе тестирование проведено для полной *(complete)* системы мобильной связи в случае аналогового радиотракта. В этом случае эталонная модель содержит 18 процессов с общим числом состояний 65. Общее число ветвей для покрытия - 183. Общий объем SDL кода 1.5 раз больше чем в первом варианте. Число входных сигналов - 36. На этот раз семантический тестер гораздо сложнее, и не только из-за увеличения числа входных сигналов, но также из-за того, что в нем были включены проверки, поступают ли все запланированные моментальные ответные сигналы системы на данный входной сигнал с формального тестера (согласно способу, описанному в 4.2). Были написаны также утверждения, контролирующие число полученных ответов. В глобальном состоянии регистрируются состояния выбранных процессов, а также значения переменных семантического тестера, существенно характеризующих состояние системы. Определение глобального состояния уточнялось во время тестирования, в конце эксперимента получено 46 тестов при общем числе глобальных состояний - 6754. Также как и в первом случае были достигнуты все выполнимые ветви в выбранных процессах. Время генерации тестов было около 50 минут на IBM PC с процессором типа Pentium. На этот раз система была гораздо лучше протестирована ручным способом. Тем не менее при автоматической генерации тестов удалось найти существенную семантическую ошибку в

системе, где из-за неправильного оператора Nextstate в диаграмме произошло неправильное разъединение вызова из публичной телефонной сети. Основную роль тут сыграло именно проверка утверждений о корректности ответных реакций. Были найдены и более мелкие ошибки в паскалевских процедурах, например, выдача неправильного номера устройства при некоторых редко употребляемых комбинациях параметров.

Результаты обоих вариантов даны в таблице 1.

Таблица 1.

	Tests	Global states
mobile-to-mobile 1	7	51
mobile-to-mobile 2	7	51
mobile-to-mobile 3	7	31
complete system	46	6754

Эксперимент полностью подтвердил применяемость описанного метода генерации тестов для телекоммуникационных систем данного размера. Практическим преимуществом описанного метода тестирования по сравнению с классическими ручным способом управляемыми тестовыми драйверами оказалось огромное количество комбинаций входных сигналов во время генерации, в том числе редко встречающихся на практике. Именно это позволяет протестировать практически все возможные ситуации в системе.

ЗАКЛЮЧЕНИЕ

В статье описан метод автоматической генерации тестов и первые его успешные применения в тестировании программного обеспечения для систем мобильной радиотелефонной связи. Приобретенный опыт позволяет утверждать, что описанный метод применим и в тестировании программного обеспечения других проектов, даже таких, размер которых гораздо больше описанного.

Авторы выражает благодарность А. Калис за постоянную помощь в реализации генератора тестов.

СПИСОК ЛИТЕРАТУРЫ

1. J.Bārzdiņš, A.Kalniņš, M.Augustons. SDL Tools for Rapid Prototyping and Testing. - In: SDL'89: The Language at Work, North- Holland, 1989, pp.127-134.

2. A.Kalniņš. Global State Based Automatic Test Generation for SDL. - In: SDL'91: Evolving Methods, North-Holland, 1991, pp. 309-312

3. G.J.Holzmann. Design and Validator of Computer Protocols. Prentice-Hall, 1991, 493p.

4. A.Ek, J.Ellsberger. A Dynamic Analysis Tool for SDL. - In: SDL'91: Evolving Methods, North-Holland, 1991, pp. 119-134.

5. A.Ek. Verifying Message Sequence Charts with the SDT Validator. - In: SDL'93: Using Objects, North-Holland, 1993, pp. 237-249.

6. B.Algayres, Y.Lejeune, F.Hugonnet, F.Hantz. The AVALON Project: A VALidatiON Environment For SDL/MSC Descriptions. - In: SDL'93: Using Objects, North-Holland, 1993, pp. 221-235.

7. A.Zaim, F.Calikoglu. Using SDL in a Commercially Available Wide Area Coverage Trunking Mobile Radio System development. - In: SDL'93: Using Objects, North-Holland, 1993, pp. 41-49.

8. A.Auziņš, J.Bārzdiņš, J.Bičevskis, K.Čerāns, A.Kalniņš. Automatic Construction of Test Sets : Theoretical Approach.- In : LNCS, Vol 502, Springer-Verlag, 1991, pp.287-360.

GRADE Version 3.0

Business Modeling Language Reference Manual

Infologistik GmbH

Munich, September 1996

Table of Contents

13 INDEX 98

1 Introduction

GRAPES-BM is a semi-formal language for modeling and simulation of business processes. It is oriented toward a detailed description of various kinds of complicated business systems: offices, information systems, production processes, enterprises etc. GRAPES-BM supports the modeling of both the dynamic behavior and organizational structure of business systems.

The application areas for GRAPES-BM are:

- Business Process Reengineering
- Analysis of workflows in business systems
- Systems analysis and requirements specification in Information System design.

GRAPES-BM supports two modes of usage:

- a semi-formal mode for modelers describing large business systems in a concise and easily readable way
- a formal executable mode for simulation of business systems in order to gather quantitative information on their behavior.

The Language Reference Manual describes the precise syntax and semantics of GRAPES-BM in its entirety, both for semiformal and formal usage.

The description of system behavior in GRAPES-BM is based on two fundamental concepts: tasks and events.

Tasks correspond to any activity performed in a business system. Large tasks are decomposed into chains of smaller ones using **Task Communication Diagrams (TCD)**. This diagram displays the business process in an easily readable flowchart type form.

Tasks have a number of formal and informal properties:

- triggering condition
- performer specification
- duration
- user defined attributes
- informal description
- objectives

and others.

All properties of a task are defined in a **Task Specification Diagram (TSD)**. The main properties of a task are visible also inside its symbol in the TCD diagram. There are two kinds of tasks: transformation tasks and decision tasks. Decision tasks describe activities with possible alternative outcomes and have **decision** symbols attached to them. Decisions may be informal or formal.

The other fundamental concept is **event**. Events represent everything (signals, information, documents, etc.)that move from task to task or influence a task in a business system. Events are represented in TCD and TSD diagrams by arrows linking task symbols with the event name adjacent to it.

There are message events, control flows and timer events in GRAPES-BM. Message events may carry data with them and this data may influence the behavior of tasks.

TCD and TSD diagrams may also contain **data store** and **data object** symbols which have informal meaning in GRAPES-BM.

The organizational structure of a business system is described in GRAPES-BM via the **ORG** diagram. This kind of diagram strongly related to the traditional ORG-chart. Though, it is more formalized in the sense that all of its elements may have formal attributes which influence the behavior of a system.

GRAPES-BM contains also some additional tables:

- ATR for describing user defined task attributes, associated with a task type
- ET for defining events (data types for message events, time moments for timer events, etc.)
- CMP for describing performer competencies
- AT for describing access to data stores
- SP for defining simulation parameters.

GRAPES-BM also uses two subtypes of DD diagram type borrowed from GRAPES/4GL.

- DD DATATYPE for describing data types of message events
- DD ER for defining entity-relationship models associated with data stores.

PD diagrams may also be used as pure comments for tasks. Other GRAPES/4GL diagram types are not used in business modeling.

The diagrams and tables describing one business model are kept together by a special "header diagram" **BM**, which has no content in and of itself in version 3.0. There may be as many task refinement levels via TSD and TCD diagrams as necessary in the given model. Tasks may be refined to whatever level of detail required by the user via TSD and TCD diagrams.

A GRAPES-BM business model may be a standalone model or alternatively one or more business models can be subordinated into a system, model by placing them under a CD diagram in a system model.

This document describes GRAPES-BM V. 3.0. . Though the main principles of GRAPES-BM V. 2.0 have been retained, the language has undergone significant changes. ORG diagrams, CMP and ATR tables have been added. TSD and TCD diagrams have retained their meaning, although a number of elements have been added to facilitate behavior description. Remote tasks have become referenced tasks, with some modifications to syntax and semantics. Referenced tasks are used now also in TSD diagrams. Existing business models in GRAPES-BM V. 2.0 may be converted to V. 3.0, with some manual adjustments due to changed semantics.

The Language Reference Manual has the following structure. First, the model tree and associated concepts are described. Then, the formal description of each diagram or table type follows. After that, some special features involving several types of diagrams are discussed. The document concludes with the summary on some auxiliary topics, such as the use of expressions.

1.1 Notational conventions

The separators shown below are used in the syntactic notation in this document. (To distinguish them from terminal symbols, they are printed in boldface.)

[]	Optional element
{}	Group of elements that can be used alternatively
|	Separator for alternative language elements
{}*	Repetition - null or multiple
{}+	Repetition - one or multiple

In the simplest cases x1, ... , xn is written instead of x {,x}* .

In some places, the standard notation traditionally present in BNF style language grammars is used also in this document:

nonterminal ::= syntax_definition

means that this nonterminal (which is a part of a larger syntax construct) is to be replaced by the given syntax definition.

Yet another convention is on lexics in GRAPES-BM.

Identifiers are strings starting with letter and containing letters, digits and underscore characters. Their length may not exceed 32 characters. GRADE editors permit the use of blanks in identifiers during input, but nevertheless they are later internally replaced by underscore characters. By default, GRADE editors also show and print these names with blanks inside. GRADE can also be configured to show names as they are stored in the repository, with underscores, using the option *Options/Settings/Underscores visible.* But all simulation oriented components show the formal names with underscores. Uppercase and lowercase letters in identifiers are treated as identical in language syntax (but they are distinguished by editors).

1.2 GRAPES-BM model tree

Just as in other GRAPES-family languages, a business model is represented as a hierarchy of diagrams and tables. This hierarchy is described via the **model tree**.

The model tree for a standalone business model has the following structure:

- the business model is headed by a special "header diagram" BM, which serves as a placeholder for the business model name;
- to the right of the BM diagram the auxiliary diagrams ORG, CMP, ET, SP can be found for that model. The ORG and CMP diagrams are described in section 2, the ET diagram - in Section 4, and SP is a special table used for simulation which is described in section 9;
- just under the BM diagram there may optionally appear ATR tables - one for each task type defined in the model. If no task types are defined, there are no ATR tables in the model tree. ATR tables are described in section 3;
- one or more DD and/or ER diagrams may be placed just under BM, for use in the Business model;
- each task has a row in the model tree. This row is started by a CM diagram, and then the TSD diagram. If the task has a refinement TCD diagram, this diagram (having the same name) is placed to the right of TSD. A task which is a part of (a refinement of) another task is placed just beneath it; and
- a task may have an AT table and PD diagram. These diagrams are placed in the far right of the row. PD diagrams are purely illustrative in BM.

Fig. 1.1 shows an example of the model tree for a standalone business model.

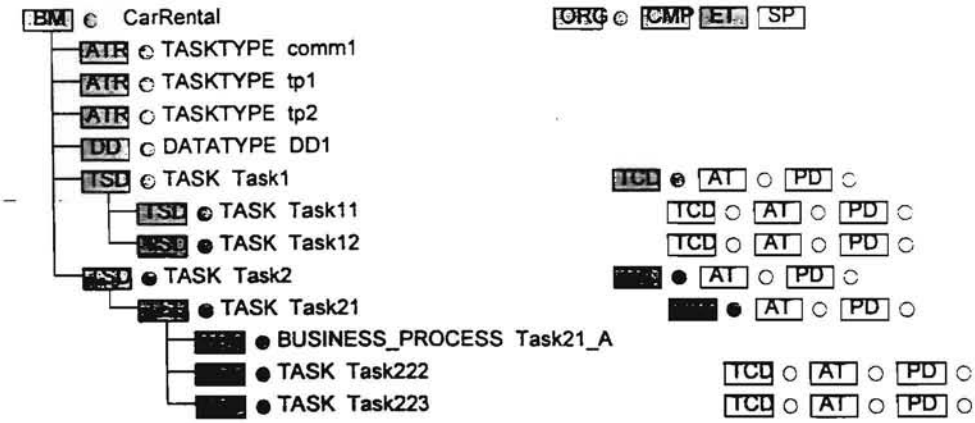


Fig. 1.1 Model tree (standalone business model)

Business modeling may also be mixed with system modeling in GRAPES/4GL . In that case:

- the business model may be placed under any CD diagram (in most cases it will be under top CD).
- the only "outer" diagrams visible inside the business model are DD and ER diagrams (they must be placed beside or above BM).
- there may be several disjointed business models in a tree. These models are independent of each other..

Fig. 1.2 shows a model tree with the business model as part of a system model.

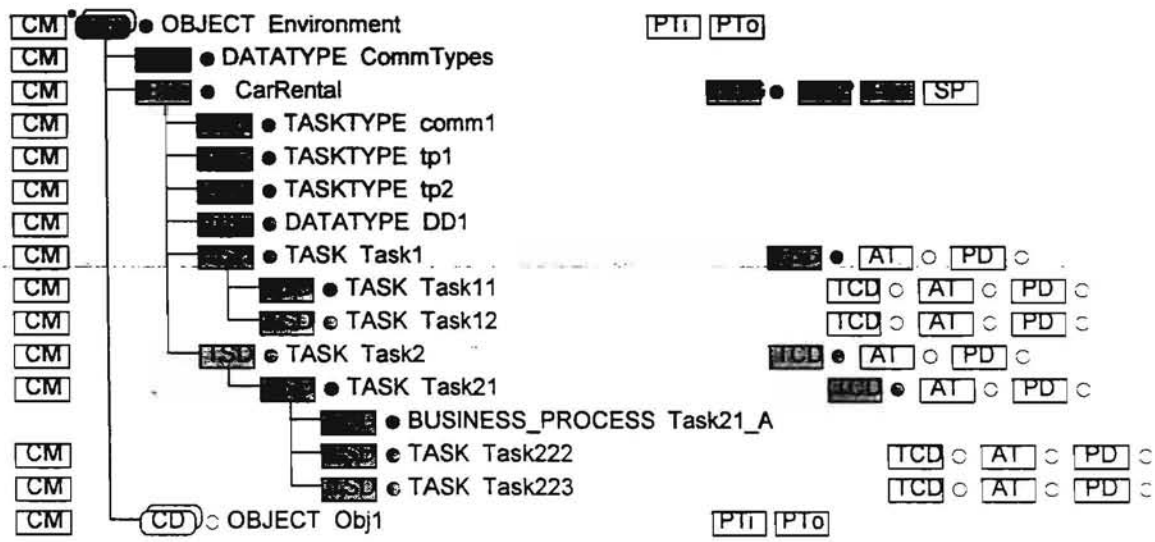


Fig. 1.2. Model tree (business model as a part of system model)

The structure of the BM model tree is as follows:

- top-level tasks which have refining TCD diagrams (referred to as **primary** tasks) are of special significance- they represent the main business functions of a system
- alternative refinements of a task via several TCD diagrams are permitted. The "main" (or the sole) TCD diagram has the same name as the task itself and is shown in the same row. Other alternative TCDs have their individual names and are placed in subsequent rows just beneath the TSD diagram.

1.3 Task visibility

An important concept in GRAPES-BM is the visibility of tasks.

The visibility rule used is the traditional one for GRAPES (as in V.2.0). Only these tasks are visible in a TCD diagram whose TSDs are :

- directly under the TSD whose refinement is being defined by the TCD (i.e., at the same level as the TCD) - the direct refinement case;

- somewhere above the given TCD (i.e.,at the same level as one of the grandparents) - refinement via a common task ;

- directly as one of the grandparents - recursive refinement, though not permitted for simulatable models.

Thus a TSD diagram is invisible if it placed in another refinement branch. If two tasks with the same name are in the same branch, only the lower one is visible. TSD names may reappear in different refinement branches - these tasks have nothing in common. Any task may have an unlimited number of occurrences in TCDs wherever it is visible, all these occurrences mean a reference to the appropriate defining TSD diagram

Normally, a task which is part of another task is placed just under it. If there is a need for common use of this task in several TCD diagrams, the task must be moved higher up in the model tree. See more on this topic in section 8.2.

All other elements of a business model - events, organizational units, task types/attributes, competencies - may only be global for the whole business model.

2 Organizational structure description

2.1 Introduction

These are completely new features not found in V2.0. They describe the organizational structure, in a broad sense, of the enterprise being modeled.

Organizational structure is described by means of two new diagrams, or more precisely, one diagram and one table:

- ORG diagram
- Competence table (CMP)

Both the ORG diagram and the CMP table are optional. ORG must be present if performers are specified in tasks. Only the elements of the ORG diagram may be used as performers in tasks for simulation purposes.

2.2 ORG diagram

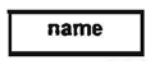
The ORG diagram is the basic facility for organizational structure description. The organizational structure is described in a tree-like manner typical to traditional ORG-charts. The main difference is in more formalized syntax and semantics. The elements of the ORG diagram may have formalized attributes. An interesting feature of the ORG diagram is the possibility to create separate subtrees within an ORG-chart as separate subordinate trees within the same ORG diagram.

An example of the ORG diagram is given in Fig. 2.1.

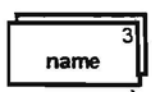
2.2.1 Elements of the ORG diagram

The following element types are present in ORG diagrams

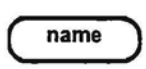
1. single organizational unit:



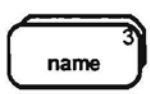
2. multiple organizational unit:



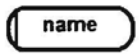
3. single position:



4. multiple position:



5. single resource:



6. multiple resource:

The informal semantics of element types is the following:

- organizational unit represents enterprise, branch, department, laboratory, etc.
- position represents any position type, like CEO, manager, programmer, secretary etc.
- resource represents any equipment or other reusable resource such as a car, computer, printer, machine, or tool.

The following relationships between elements are present in ORG diagram.

- *consists of*
- *owns*

Elements may follow each other according to the following rules:

1, 2	may be followed by 1, 2, 3, 4	via *consists of*
1, 2, 3, 4	may be followed by 5,6	via *owns*
5, 6	may be followed by 5, 6	via *consists of*

It means, that both single and multiple element of the same kind may have the same relationships.

The same graphical notation - a line from an element to its follower is used to represent all types of relationships. This is due to the fact that the relationship type is determined uniquely by the types of the source and sink elements of the line.

The requirement that elements of ORG have a certain hierarchy is based on specified rules, which are essential for the informal semantics of the diagram and its readability. The violation of these rules does not affect simulation. Therefore no syntax checks are performed to ensure that elements follow each other correctly in the current version.

ORG diagrams may contain the standard free comment symbol which does not affect semantics.

2.2.2 Attributes of ORG elements

Name is the identifier of each element in an ORG diagram. Aside from that, any element of ORG diagram may have the following optional attributes:

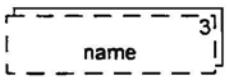
- **type** - *internal* or *external*, *internal* is by default, *external* specifies that the organizational units belongs to an external partner of the enterprise;
- **number** of **instances** (for multiple ones),

- **availability;**
- **cost per hour;**
- **competence** (identifier, one or several, comma separated, i.e., actually a competence list can be used);
- **efficiency level.** It can be any real number, e.g., 0.5, greater than 0. In this case it means that given performer can do his work with efficiency 50%. Duration time for the task corresponds to the efficiency level 1. The real duration can be obtained via the formula

  formal_duration / efficiency_level.

  If a task has several ANDed performers/resources in its performer expression in a TSD, the minimal efficiency_level (from the referenced ones) is used in the formula.
- **employee name**- only for position type elements.

The default setting for the **Type** Attribute is that for internal actors. The External type attribute is shown by the dashed line used for the element contour, e.g.,



Normally the usage of external organizational units (and their "components") as performers should make the task an external one, but formally, task externality is independent of performer externality.

Number of instances may be used only for multiple elements. If it is not specified, an unlimited number of instances is assumed. The number of instances must be a non-negative integer constant. Zero means the performer is unavailable, one is the same as a single performer.

Availability specifies the time intervals when the ORG element is available as a performer. Availability is defined as a sequence of time unit specifications in a descending order. It includes years, months, days, hours and minutes. Two abbreviated formats may also be used:

- from years to days (date part only)
- only hours and minutes (hour part only)

Day specification (a day constant) may be in one of two forms

- date of month (from 01 to 31)
- weekday (from MON to SUN)

Year specification (year constants) are four-digit numbers in the form 19xx or 20xx, month specifications are from 01 to 12.

Each year, month and day specifications may be

- an asterisk character (not in parenthesis!)
- a single constant value (in parenthesis or not)
- a single interval i.e. constant-constant (in parenthesis)
- a comma-separated list of constants and/or intervals (in parenthesis)

The separator between the date units is a period. The date and hour parts are separated by just one blank space (if both parts are present).

The hour part (if present) is always placed in parenthesis. Any hour constant contains combined hour and minute notations, separated by a colon, i.e. hh:mm, where hh is from 00 to 23 and mm from 00 to 59.

The hour part may be:

- a single constant
- an interval , i.e. constant-constant
- a comma separated list of constants and/or intervals

The whole availability specification may contain no additional blank spaces (except just one blank space between date and hour parts). The whole specification is contained in double-quotes. All numbers always contain two or four digits respectively.

The ends of an interval must be in increasing order. It is not permitted to mix the date of month and weekday specifications in one availability definition. If an interval including the end of the month is to be specified, it must be split into two intervals, e.g. (01-03,25-31). Invalid dates such as 02.30 never make the ORG element available, although no error is reported.

Examples of availability specifications:

```
"(08:00-17:30)"
"*.*.(05-20) (09:00-18:00)"
"*.*.(MON-FRI) (08:30-16:15)"
"*.*.(MON-FRI) (09:00-13:00,4:00-18:00)"
"*.(04-09).(MON-FRI)"
"*.*.(MON,WED) (10:00-14:00)"
"(1995-1997).(05,09).01 (09:00-16:00)"
"*.*.01 (00:00-23:59)"  - available on the first of every month, 00-24 (24 is never used!)
"*.*.01"  - the same as previous
```

The semantics for availability are the natural ones. * means no restrictions on the unit. All intervals and value lists with different units are always combined together (e.g., working hours are applied to all specified working days). If the lowest unit has a single constant value, the availability is valid while this unit has the specified value. Omitting the date part means availability every day. For example, "(17:30)" means availability for one minute every day at 17:30 (a very strange performer, certainly, a more realistic case is availability for just one day every month).

Cost per hour is an integer or float constant. It is used to calculate automatically the cost of a task, by multiplying it with a task's duration.

The **Competence** list of an ORG element specifies its competencies (from the CMP table, see 2.3). Competence is a performer characteristic in a broad sense, which may be used in task specifications to select a performer with the given characteristics. It is mostly used with positions.

Employee name may be used only for single positions. If there are several similar positions distinguished only by employee names, the position symbol has to be repeated the required number of times. Employee name must be used as an identifier in GRAPES-BM, since it is the only way (aside from competencies) to select one specific performer (in a task's performer selection expression) from many with the same position name.

There are two display modes for the ORG diagram: the short and the long display forms. In the short form no attributes of elements are visible in the diagram, except number of instances and externality (which are always visible). In the long form all attributes (which have been defined by the user) are visible inside all elements of ORG. The same two modes exist for hardcopy printing.

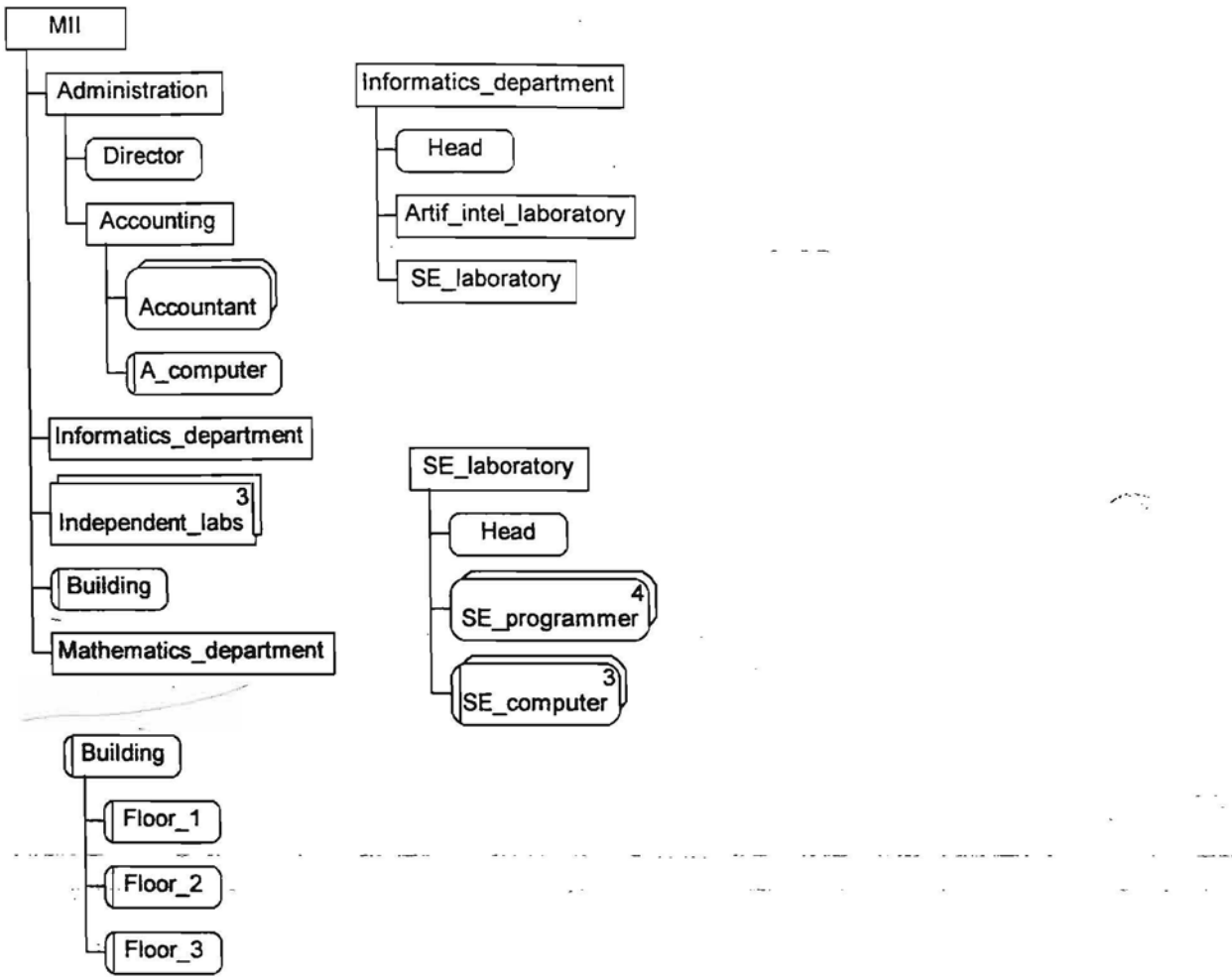


Fig. 2.1 Example of ORG Diagram (in the short mode)

2.2.3 General structure of ORG diagram

In general, the ORG diagram appears as a forest containing several trees. The nodes of the trees are the above mentioned elements, and branches represent the above mentioned relations between the element instances (in a normal way, from top to down).

A typical situation in ORG diagrams, is such that the leaves of a tree are refined further by other separate trees in the diagram. The reference is made by virtue of the fact that the leaf has the same name (and element type) as the root of the refining tree.

The names of separate objects (i.e., the names of tree roots) must be unique diagram-wide. Names of non-root objects may repeat. A tree may be referenced several times, each reference denotes a separate object (with the same characteristics). Any of the elements may serve as a tree root. There may be several unreferenced roots.

Another requirement is that units and resources directly under a common parent must have unique names.

There is an exception for positions. Position is the only performer type, where several equally named objects may be placed at the same non-top level (with different competencies or employee names, as a

rule). If such a feature is used, then none of these equally named position elements may have a resource symbol under it.

By default, the ORG diagram editor is in the "vertical refinement" mode, as shown in Fig. 2.1 However, it is possible at every node (independent of other nodes) to switch to the horizontal layout for successor nodes, then switch back to the vertical layout at some lower point and so on. In this way the example from Fig. 2.1 may be displayed as depicted in Fig. 2.2 Thus one can obtain also the traditional ORG-Chart form for an ORG diagram.

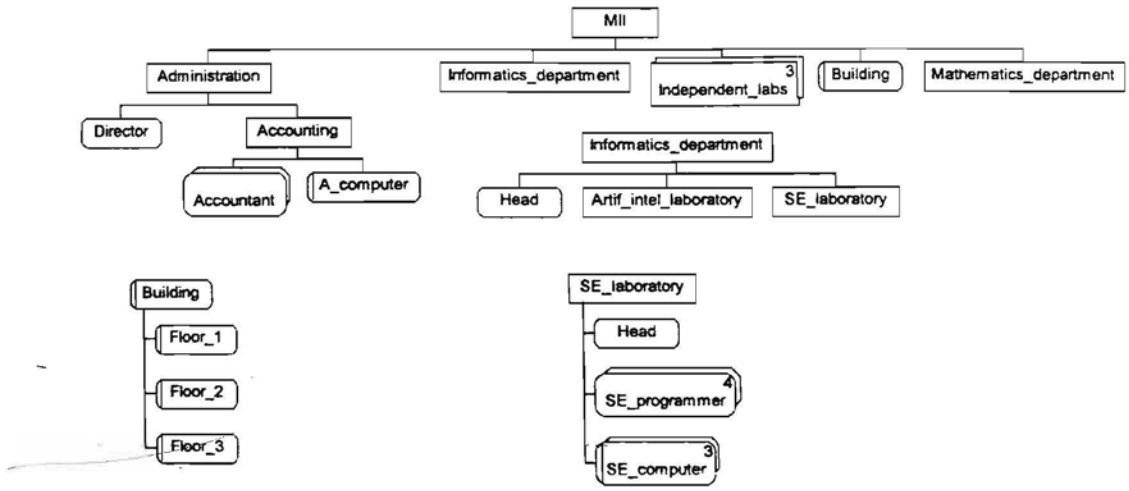


Fig. 2.2 Example of Horizontal ORG diagrams

2.2.4 The formal semantics of ORG diagrams

Since the ORG diagram is a set of pure trees, every occurrence of an element (unit, position, resource) with a given name defines a real separate element, which can be used as a performer. Two equally named elements in a tree represent two performers, which can be distinguished by using qualifiers in performer selection expressions.

Any independent tree, whose root is referenced nowhere, represents an independent performer (of the appropriate type). However, as soon as a reference to it (i.e., an equally named leaf in another tree) appears, the performer is placed in the referencing tree and there is no more independent copy of it. Thus in the ORG example (Fig. 2.1 or Fig. 2.2) there is only one instance of *Building* or *SE-laboratory* available during execution. If the root name occurs in a tree in a non-leaf position (i.e., with a subtree beneath it), then this occurrence is marked as an error. In other words, it is forbidden to make a local redefinition of an independent tree.

The same *consists of* (or *owns*) semantics is assumed for descendants of multiple elements. It means that there are several elements with the same internal structure or ownership, e.g., 3 (unnamed) laboratories containing the same set of positions, 4 programmers each owning a personal computer (with equal characteristics) etc.

In addition, inheritance is applicable to availability and efficiency. If at any level (starting from the top one) one of these attributes is specified, and is not specified in a subsequent level (or levels), then the value is copied to this subsequent level (or levels). All element types are equal from the inheritance point of view, only the tree structure is significant. When at a lower level the attribute is again defined, it redefines the inherited value. Both for availability and efficiency it means complete redefinition.

Cost and competence are not inherited, since they may have different meanings at different levels.

The general default for attributes (when nothing is inherited) is always for availability, 1 for efficiency, none for competence and 0 for cost.

If an ORG-unit cost per hour is required, then either the explicitly specified value is taken, or the sum of components (just one level below) cost per hour is taken (if the explicit value is absent).

2.3 Competence table

Competence table (CMP) defines the list of competencies which can be used in the business model. Each competence definition entry contains its sole attribute - the competence identifier.

All table entries may contain a comment.

The competence identifier can be an arbitrary identifier with no additional semantic meaning. All competence identifiers should be unique. Its sole purpose is in enabling a task's performer expression to select one from a group of similar performers based on a specifically defined competence. Competence is mainly used for positions, where it can help to select one from several equally named positions at the same level. Nevertheless, competencies may be used to give a certain object-oriented style to the ORG diagram. They may be used to model subtype/supertype relations between ORG objects as well as "deputy" performer type relationship. Competencies also permit the user to select performers for tasks based on qualities independent of their placement within the organizational structure.

3 User defined Task Types and User Defined Attributes for Tasks

These are completely new features in GRAPES-BM, not found in V2.0. Any Business Model may contain tasks of different types. Each task type has its own list of attributes.

A task type is defined by the corresponding attribute table (ATR), which contains the type name. Untyped (tasks without assignment of a task type) tasks have no attributes.

Additionally, task types can be defined without defining attributes by leaving the ATR table blank.

Example of Attribute Table is given in Fig. 3-1.

Name	Type	Default	Unit	Formula
cost_s	integer	10	DM	
people	integer	1		
man_hour	duration			duration*people
name	string			
redo	float	0	%	
days	list	1,3,5		
total_costs	integer			cost_s*people

Fig. 3-1. Example of Attribute Table ATR

The Attribute Table contains the following columns:

Name defines the name of an attribute. It is an identifier. All names within a table must be unique.

Type defines the data type of the attribute. Only the following elementary types can be used:

integer, float; string, time, duration.

An additional special type *LIST* is permitted, in this case the value of the attribute may be any list of elementary data elements. String and LIST may not be used for derived attributes, nor may they be referenced in formulas. The default maximum length for strings is 255. Neither the additional type attributes (as found in GRAPES/4GL) nor user defined "elementary types" may be used here.

The sole use of string attributes in this version is to allow the user to define informal characteristics for a task for later export to other tools (e.g. "workflow" systems).

Default defines the default value of the attribute. A proper constant (i.e., literal) of the appropriate type or a named constant from SP may be used here. The default value is used if the attribute is not re-defined in the TSD where it is used.

Unit defines a text string which defines the units of measurement for the attribute (typically: DM, USD, %). The unit has no effect on attribute value, e.g., 70% is treated as integer 70 in formulas. The unit is a pure comment and does not appear outside the ATR table.

Formula defines the value of derived attributes. It is an arithmetic expression composed of other attributes and constants. The formula may be redefined in a TSD for an individual task, and input event fields may also be referenced there. If the type is meant for elementary tasks, the formula may contain only standard arithmetic operations. If the task type is meant for transaction (non-elementary) tasks (see more in section 7.4), the formula may contain also so called vertical operations (SUM, MAX, MIN, AVG). The operand in such operations may be any numeric or DURATION type attribute in any ATR table. The semantics of such operations are explained in section 7.4. Random generator functions of the appropriate type may also be used. Named constants from SP may also be used. See more on formula syntax in section 10.2.

Formulas may also contain the two predefined attributes of the task: duration and cost (the duration reference in formula of Fig. 3-1 means the task duration, which is of type duration).

A derived attribute may reference another derived attribute in its formula, but no circular references are permitted. The actual reference validation is performed in the TSD during simulation where redefined attributes are also taken into account. The order of the entries in the ATR table has no semantic meaning.

Numeric constants are defined in the standard syntax.

Time constants have either the full date_time form or date only; standard separators are used, e.g., "1995.05.19 18:38:00"(seconds must be present in the time part). Duration constants may contain any consecutive units from **days** to **seconds** in descending order (separated by colon), e.g., "100d:5h", "1h:30m:53s", "95m". The numeric elements in duration constants may be integers or floats, e.g., "0.5h" is also permitted. See more on constant syntax in section 10.1.

A task of a particular task type may possess all the attributes which are defined in the corresponding ATR table for this task type. For each task of a given type, some attributes may be redefined in the TSD of a particular task. The definitions of the remaining attributes are taken from the ATR table. If both a default value and formula are defined, the formula takes precedence. If both columns default and the formula in the ATR are empty for an attribute and it is not redefined in the TSD, the attribute has undefined (NULL) value (and consequently, is ignored in any statistics computation).

Several ATR tables may contain equally named attributes. No restrictions apply unless such an attribute is used in vertical operations. In the latter case the attribute must have the same type in all ATR tables where it appears.

4 Event table

4.1 General structure

The **event table** (ET) describes all events appearing in a business model. There is only one event table per business model. In the model tree, ET is located in the "header" row of a business model, i.e. to the right of BM. Fig. 4.1 shows an example of the event table. The event table has the following columns:

- name
- category
- type
- Persistence interval
- Transfer time
- Description

Name	Category	Type	Persistence interval	Transfer time	Description
Application Car	message material	ApplDT		"5h" "30m"	
Morning	timer	time ("*.*.* 09:00")			
Earthquake	message		"5m"	exponential ("3m")	
Registration	complex	(Application, Car)			
From5to6pm	timer	time ("*.*.* 17:00")	"1h"		
Every2m	timer	REPETITION (exponential ("2m"))			

Fig. 4.1 Example of Event Table

The **name** of an event is an identifier. Any event is visible in the whole business model.

Category has one of the predefined values:

- message - any event carrying some information
- material - an event carrying some physical objects
- timer - timer event
- complex - complex event containing several elementary events

or a user defined category name, which can be any identifier. There is no formal difference between *message*, *material* and any user defined category since they are treated informally in simulation and are meant only to enhance the understandability of a business system. Events of any of these categories may have a data type specified and, consequently, carry data of the specified type. If the type is not specified, then the event carries no data (it is like a Signal). All these categories are sometimes informally referred to as message events.

Timer and complex events are of a completely different nature, and they are described in 4.2 and 4.3 respectively.

Pure **control flows** have no name and, therefore, don't appear in ET at all.

Type is a predefined or user defined type name. It specifies the data carried by the event. Type must be either elementary (from the predefined list: *integer*, *float*, *duration*, *time*, *string*) or a user defined record. This record may contain other records in turn (in arbitrary depth). In general, the types of record fields may be arbitrary. But the subset of record fields really used in the data context of business modeling (i.e. referenced as input or output event fields in attribute sections, SET-options, REPEAT-options, triggering conditions or decisions) must satisfy a stronger requirement. These fields must be:

- elementary
- have one of the types *integer*, *float*, *duration*, *time*, without any additional type attributes.

Events of the type *string* are also not permitted to be referenced in the above-mentioned data context. The type validity of each event or event field to be used in the context of data is checked at the point of its use in the TSD.

For timers the same *type* column is used for time moment definition, and for complex events it is used for component definition.

Persistence interval column is either empty or contains a duration constant

The duration constant may be a proper constant or a so called constant duration expression. This is a duration expression which may contain random duration functions and permitted operations, but with proper constants as the only ultimate arguments. See more on precise syntax in section 10. Named constant from SP may also be used for persistence interval.

The persistence interval characterizes the time period for which the event persists in an input event queue. If the interval is not specified, the default persistence is

- 0s for timers
- unlimited time interval for any other event

See more on actual event semantics in 4.4.

Transfer time: this is a duration constant or a constant duration expression (the same as for persistence) which defines the time necessary to transfer the given event between the sending and receiving task. The transfer time can be redefined in the TCD diagram (separately for each occurrence of this event). Named constants from SP also may be referenced, either directly or in random expressions. See more on syntax in section 10.

Transfer time is ignored for timer events. For an event route consisting of several arrows (see 6.5) transfer time is counted once.

4.2 Timer definitions

Timers are specified by entering their definition in the *type* column.

Timer definition syntax is in one of the two forms:

- time form
- repetition form

The **time** form has the syntax:

TIME (time_specification)

where time_specification has a similar syntax to the availability definition in ORG diagrams.

More precisely, time specification is a sequence of time unit specifications in descending order. It includes years, months, days, hours and minutes. The sequence always starts from years and may end at any of the units. The only exception is that hours and minutes are always combined - if hours are present, then minutes also must be.

Day specification (day constant) may be in one of two alternative forms:

- day of month (from 01 to 31)

- weekday (from MON to SUN)

Year specifications (year constants) are four digit numbers in form 19xx or 20xx, month specifications are two digit numbers from 01 to 12, hour specifications - from 00 to 23 respectively, minute specifications - from 00 to 59.

The separator between years and months and between months and days is a period, between days and hours - exactly one blank space, between hours and minutes - a colon.

Each of the year, month and day specifications may be

- a single constant value (in parenthesis or not)

- an asterisk character (not in parenthesis)

- a single interval, i.e. constant-constant, in parenthesis

- a comma-separated list of constants and/or intervals, in parenthesis

Hour and minute constants always appear combined. Thus, hour-minute specification may be

- a single constant in the form hh:mm (with or without parenthesis)

- asterisk either in the hour position, or in the minute position, or in both (*:mm, hh:*, *:*), with or without parenthesis

- a single interval, i.e. constant-constant, in parenthesis

- a comma separated list of constants and/or intervals, in parenthesis

The whole time specification expression may contain no additional blank spaces (except the one between days and hours). The whole specification is embraced by double quotes. Numeric values for month, day, hour, minute must always contain exactly two digits.

Four asterisks may be used instead of one in year position, and two asterisks may be used in month, day, hour and minute positions. The semantics is the same, e.g., "****.**.**" is the same as "*.*.*". Mixing asterisks with digits is prohibited.

The ends of an interval must be in increasing order. All values in one list, i.e., both interval ends and single constants must be in strictly increasing order. If an interval includes e.g., the end of the month, it must be split into two, for example

"*.*.(01-05,15-16,28-31) 09:00"

The remark refers also to weekdays, e.g.,

(MON-TUE,FRI-SUN)

for service closed on Wednesday and Thursday. Invalid dates such as 02.30 or 11.31 simply never occur.

Valid time specification examples:

```
"*.*.* 09:00"
"*.*.(MON-FRI) 17:00"                    /* once per workday at 5PM*/
"*.*.(MON-FRI) (09:00-17:00)"            /*  once per minute during the business day */
"*.(04-09).(TUE-SUN) (09:00,10:00,11:00,12:00,13:00,14:00,15:00,16:00)"  /* once per hour, in
                     summer daytime, except Mondays */
"1996.*.04"                              /* on the fourth of every month in 1996 at 00:00 */
"*.(03-08)"                              /* on the first of the specified months */
"*.*.* *:15"                             /* quarter past every hour */
```

The special built-in function **Start_time** may also be used as a time_specification. Such a timer is activated only once at the start of the simulation session. The syntax is:

TIME (Start_time).

A time-based timer is triggered every instant the smallest time unit explicitly specified in the time specification becomes valid (but the periodicity of it determined by the lowest interval or * element).

The **repetition** form has the syntax

REPETITION (duration expression).

The duration expression may contain only duration constants and functions of constants, i.e. the same style constant expression with random functions as those used in the *transfer time* column. See more on the syntax of such expressions in section 10.

Examples.

REPETITION("10m:30s")

REPETITION(UNIFORM("1d","3d"))

Repetition-based timers are triggered periodically after the specified interval (constant or random) has elapsed. The first triggering occurs after the specified interval has elapsed from the session start-up time.

If more complicated behavior in terms of time is required (e.g., for a realistic simulation load generator), timers must be combined with availability.

4.3 Complex events

Complex events have their own category. Their definition contains a comma separated list of elementary events in *Type* column. Complex events in this version may not be directly sent or received by elementary tasks. They may be only used for bundling together several related events having the same route in some high level TCD diagrams. They must be finally refined into elementary events at some TCD level (using syntax defined in 6.2.12). No nesting of complex events is permitted.

Transfer time and persistence is ignored for complex events. Timers may not be used as elements of complex events.

4.4 The semantic aspects of event behavior

All events are created by tasks or timers and sent to their destination queues at other tasks. Then they are taken from these queues and used for triggering those destination tasks (they are "consumed").

If considered in detail, there are two types of event behavior in queues:

- **lasting** semantics
- **enabling** semantics.

For any category except timer, the "lasting" semantics is assumed. If the persistence column is empty, it means "lasting forever", namely the event enters the target queue and remains there until it is consumed by the task; there is no difference whether the event alone triggers the task or several ANDed events trigger the task. Lasting semantics is used also for control flows not definable in ET. "Enabling semantics" is never used for non-timer events.

If the persistence column contains a duration constant, it means "lasting for this duration", i.e., after the expiration of this period the event vanishes from the queue.

On the contrary, for timers the "enabling semantics" always is assumed. An empty persistence column for timers means enabling ("0s") semantics. Explicit duration value d in this column means enabling (d) for timers. The "enabling" semantics is the following. First, such an event persists in the queue only for the specified time interval, after that it simply vanishes from the queue. In particular, enabling ("0s") (which is default for timers) means that event must be used at the same system time moment when it appears in the queue (a lot of actions may occur at one system time moment, e.g., several task instances may start). Such an event vanishes from queue when the system time is advanced.

Second, enabling semantics is different from a consumption point of view. There is no difference when the event alone triggers a task - it is immediately consumed from the queue after the triggering. However, when an enabling event triggers a task in an ANDed combination, with one or more lasting events, the enabling event remains in the queue (the lasting ones are consumed). Thus an enabling event can "pair" with several sets of lasting events while it persists in queue. For example, if a task is triggered by *e AND t*, where *e* is a lasting event and *t* a timer with default persistence, then one instance of *t* pairs with all instances of *e* which are present in the queue when *t* arrives, and the corresponding number of task instances are started (provided that there is sufficient number of performers available). Only the expiration of the time interval deletes the enabling event from queue in such a situation.. Sometimes in rare instances, several enabling events are ANDed (i.e. two timer events, one an interval the second a frequency are used together), but note that the enabling events are all consumed at triggering in this case.

The consumption of timers depends on the exact set of events used for this particular triggering. When there are ORs in the triggering condition involving timers, such as the expression *(tim1 AND e1) OR tim2* where *tim1* and *tim2* both are timers, this should be taken into account (*tim2* is consumed when it triggers but tim1 is not).

Normally there is either zero or one timer in an input queue. However, if a persistence interval exceeds repetition period (this may happen for random intervals) there may be more than one timer in the corresponding queue. From the enabling point of view it makes no difference how many timers are in the queue (if there is at least one), their consuming is as for other events.

5 Task Specification Diagram

5.1 General form and role of TSD

Any task used somewhere in a business model has a **Task Specification diagram (TSD)**.TSD diagram describes all properties of a task and its possible links to other tasks (its neighborhood).

Formally, the TSD diagram is the definition of the task. All properties defined there apply to all occurrences of this task in any of the task communication diagrams. The neighborhood description of a task in its TSD is made up from all of its occurrences in TCD diagrams.

A task is called **elementary** if it has no further refinement via TCD diagram, and a task is called **complex** if it has at least one refinement via TCD. For an elementary task all of its properties are defined in its TSD. Properties of a complex task are to a great deal redefined by its refinement. Complex tasks actually correspond to transactions (see section 7).

From the functionality point of view there are **transformation tasks** and **decision** tasks. Transformation tasks have only one possible continuation, while decision tasks have several possible continuations from which one or more is selected. Decision tasks are distinguished by the presence of **decision symbols**. Transformation and decision tasks are not subtypes of TSD, these are just subclasses of the same diagram type.

The properties of a task (its triggering condition, performer expression, attributes, etc.) are described in the central symbol of a TSD diagram - in the **task body**. For decision tasks the body is followed by **decision symbols**.

The neighborhood description consists of **input events** and **output events**, represented by arrows. Input events go from **referenced task** symbols (or **referenced timer** symbols) to the task body. Output events go from the body (for transformation tasks) or from the decision symbols (for decision tasks) to referenced task symbols.

Referenced task symbols correspond to the neighbors of the task as they appear in a TCD. Therefore each referenced task symbol contains the name of the corresponding neighbor. On the other hand, referenced task symbols appear the same way once more in the refinement TCD of this task. Thus these symbols provide additional linkage between two adjacent levels of TCD diagrams (i.e., they improve linkage readability and resolve ambiguities of linkage based solely on event names). Referenced task symbols are used in representing both incoming and outgoing links of a task.

Formally the TSD diagram consists of:

- body (always one);
- decisions symbols (if the task is a decision task);
- arrows for input/output events with event names assigned to them;
- referenced task (timer) symbols associated with input/output arrows, containing task names (or name lists);
- textual detailing which may be placed on input/output event arrows beside their names to provide additional details on event receiving/sending;

- data store symbols representing both informal or formal data stores or material stores. Symbols are linked to body via possibly named access paths which represent data flows. The characteristics of these flows are specified in AT tables.
- data object symbols linked to body via the same access paths

Fig 5.1 shows an example of simple TSD (a decision task case)

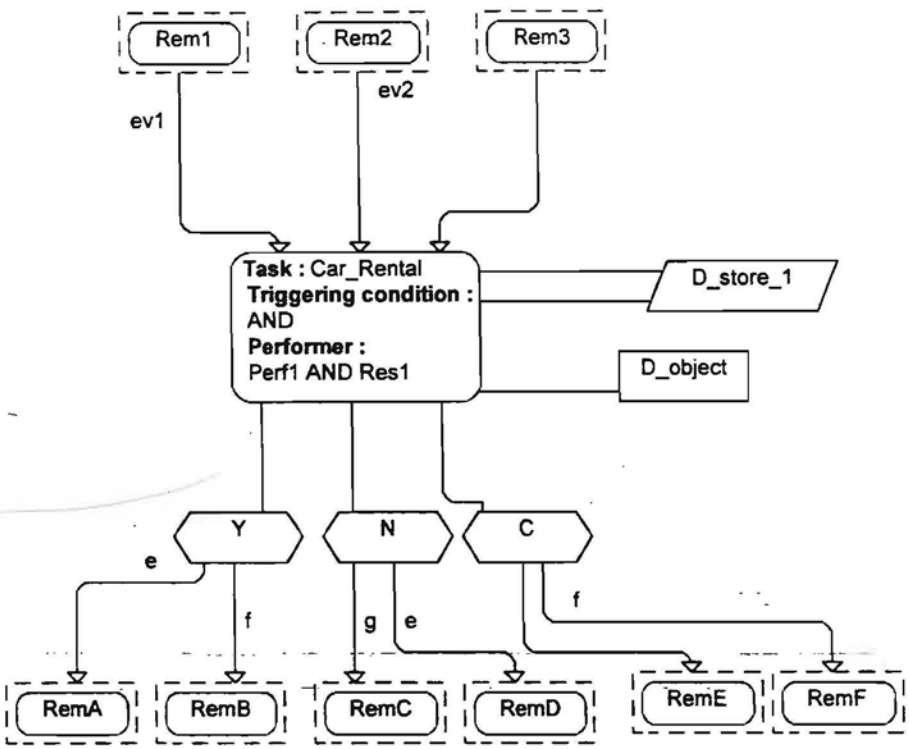


Fig. 5.1 Example of simple TSD

There are some formal rules for TSD diagram structure:

- for decision tasks, output events may start only from decision symbols, but not from the body
- each referenced task (or referenced timer) symbol is associated with only one (input or output) event arrow
- the referenced timer symbol may be associated only with input event arrow.

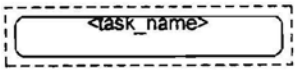
Though the TSD is the formal definition of a task, often only TCD diagrams are manually built by the user and TSD diagrams are generated automatically by GRADE editors (see sec. 6.6 and GRADE on-line help). If only those task properties are used which are visible in the TCD diagram, then TSD diagrams need no manual modification and there is complete consistence between TSD and TCD. However, if a TSD diagram (its body or decisions) are in fact manually updated, then it becomes the primary definition of task properties, and not the TCD diagram.

In comparison with version 2.0, the TSD features have been significantly extended. Referenced task symbols have been added and therefore referenced task symbols contain no names after conversion from V.2.0. Some other manual updates may also be necessary.

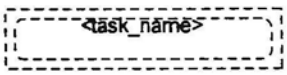
In the next sections all elements of the TSD will be explained in detail.

5.2 Referenced task symbols

The referenced task symbol in a TSD diagram represents a neighbor of the given task when one or more occurrences of this task appear in some TCD. There are three kinds of referenced symbols:



referenced internal task

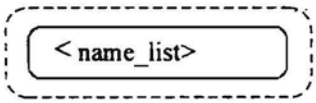


referenced external task

referenced timer

The three kinds of referenced symbols represent neighbors, either internal or external tasks respectively, or incoming timers. Referenced internal and external tasks may represent both incoming or outgoing links of a task, referenced timer may be only incoming. The referenced internal and external task symbols are distinguished only for better readability, from the formal semantics point of view they are equal.

In general, both for internal and external referenced tasks, a name list (comma separated) may be used instead of single name:



For referenced external task the name list may also be empty (but not for an internal one). Such an empty list corresponds to a neighbor which is an unnamed external task.

The main association between a task's definition in its TSD and its occurrence in a TCD (and between TSD and refinement TCD) is still via event names. Referenced task symbols with their names only help to detail this association.

Referenced tasks associated with incoming events are also called incoming referenced tasks, and the same applies to outgoing ones.

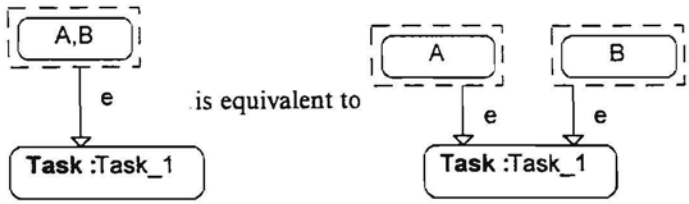
Normally there are as many incoming referenced tasks (of the appropriate kind) and respective incoming event arrows in a TSD as there are different events entering this task in a TCD. If in a TCD, several equally named events enter the task from different tasks, then the names of all of these neighbor tasks are grouped together in one referenced task symbol, associated with the given incoming event. All incoming control flows normally also are represented by one referenced task symbol with an appropriate name list, associated with the unnamed input.

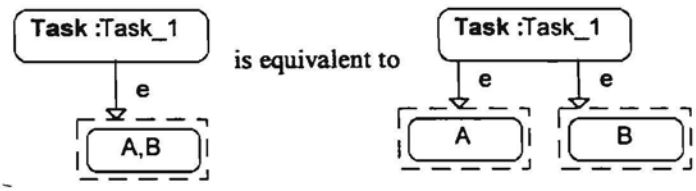
The situation is similar for outgoing events. Different events leaving the task (or one of the decisions in case of a decision task) correspond to an equal number of outgoing referenced tasks. If several equally named events leave the same source, their destination names are placed in one referenced task symbol.

This, in general, is the default appearance of a TSD when automatically created by GRADE editors. However, it makes no-difference whether referenced task names are grouped in one list or they are placed

separately in several referenced task symbols. Formally, when a name list is used in a referenced task, it is completely equivalent to several referenced tasks connected to the same partner.

Namely,





It is permitted to combine in one referenced task's name list, names of both internal and external tasks.

But the default editor principle in the case of many equally named events is to place internal neighbor task names in one internal referenced task and external neighbor task names in another external referenced task. From the formal syntax point of view, grouping in name lists is completely irrelevant.

There is a mandatory requirement, that if there are several events with the same name (in particular, control flows) then the name lists within the corresponding referenced tasks must have no common elements. External referenced tasks may also have an empty name list if the relevant neighbor is an unnamed external task (but this "empty name" may be simply ignored, if the event comes/goes also from another named task). If a neighbor of a task in a TCD already is a referenced task, it is represented in the TSD by the same referenced symbol (the name (or name list) is also retained).

Each incoming timer is represented by a separate referenced timer symbol.

If a task has several occurrences (in one or several TCD diagrams), the referenced task name lists are summed up from all occurrences.

Referenced task names in TSDs are semantically insignificant for incoming named events when a task is elementary. In any case there is one queue for a named event (but there may be also no queue at all for this event at a given task occurrence when there are no connections, see more in 6.5). For control flows referenced names have some significance since there is potentially one queue for each different referenced task name.

But for outgoing events, the referenced names are always the basis for event routing (see 6.5).

In any case, there should be no superfluous referenced task names in any TSD, except the natural name lists appearing in case of several occurrences of the task. Superfluous referenced task names may cause problems for simulation, see sec. 6.6. There are special GRADE editor facilities for removing superfluous referenced task names (i.e., those not used in any TCDs).

On the other hand, all neighbor names from any of the task's occurrences must appear in a relevant referenced task symbol.

The special case is the TSD for primary (i.e. top level) tasks. Though there is no TCD above, such a TSD may contain a referenced task symbol naming another primary task. This option is used to ensure the exchange of events between primary tasks (see more in 8.1).

5.3 General contents of the task body

The task body is the main element of the TSD, where all task properties are described, in separate sections.

The following sections are available

- Task type
- Triggering condition
- Performer expression
- Informal description
- Objectives
- Constraints
- Execution mode
- Priority
- Duration
- Max instances
- Attributes
- Alternatives

Fig. 5.2 shows an example of a task body where all sections are present. All sections are optional, and there may be a task , where only its name is present in its body. This name is always the task name itself.

The most significant (and most used) sections from the behavior description point of view are triggering conditions, duration and performer expression.

These sections are described as the first ones.

Task : task_name **Type :** Type_name
Triggering condition :
ev1 AND ev2
Performer :
(Perf1 AND Res1) OR
(Perf2 AND Perf3 AND Res2)
Informal description :
Any text
Objectives :
For something
Constraints :
This must be
Execution mode : Manual
Duration : "3h"
Max instances : 7
Priority : 0
Attributes :
Redo_probability: 30;
Personnel_costs: 23.5*ev2.costs;
Process_costs: COST*ev1.length
Alternatives :
A1: PROBABILITY=70 %
A2: PROBABILITY=30 %

Fig. 5-2 Task body in TSD

When a task appears in a TCD as an occurrence, the following sections of the body may also be present there:

- task name
- triggering condition
- performer expression
- task duration specification.

In a correct model, any task information included in a TCD must contain the same information as the corresponding section in this task's TSD, or be empty. GRADE editors provide support in maintaining this data consistency since sections from the TCD are automatically transferred to TSD when a TCD is modified. If the information in the TSD contradicts the corresponding section in the TCD (usually the result of a manual modification to the TSD by the user), then in simulation the TSD information is used.

In addition, tasks in a TCD may contain sections which never appear in a TSD

- occurrence comment
- transaction control options (START, NOSTART, END) (see more in sec. 7)

In a TSD, the informal description already plays the role of a comment, therefore there is no special comment section.

The following sections of the TSD are informal in character:

- informal description
- objectives
- constraints
- execution mode

5.4 Triggering condition

Triggering condition describes which events or event combinations must have arrived via incoming event arrows from other tasks and, consequently, must be in the event queues of a task in order for this task to start. When the task actually starts, the events which have triggered the task are removed from the queues. This is known as the **consuming** of an event combination.

Triggering condition is significant only for elementary tasks.

5.4.1 Simple cases

The simplest form of triggering condition is that consisting of just one keyword

AND or

OR

AND means ANDing together all possible input events (which are present in the TSD). namely, one instance of each distinct input event is required. If several equally named event arrows (i.e., from different neighbors) enter a task, only one instance of such event is required. OR requires any one of the input events to be present. Only this one event is consumed when the task starts.

The Triggering condition may be completely absent as well. In this case the default simple AND is assumed.

If only one input event enters the task, that event name also may be used as the triggering condition. This again means the same thing - one instance of the event must be present.

5.4.2 Syntax in general

The general form for the triggering condition is a special Boolean expression using incoming event names. This expression may be:

- a standard Boolean expression, using AND (high priority), OR (low priority) and parentheses from incoming event names;
- special_AND_expression;
- OR_expression built from special_AND_expressions and standard Boolean expressions.

A special_AND_expression is an and_list associated with one or more additional statements such as WHERE, AND ALL, <integer> event_name, etc. Formally, the special AND expression is one of:

- (and_list WHERE condition)
- (and_list AND ALL event_name)
- (and_list AND ALL event_name WHERE condition)
- (<integer>event_name)
- (and_list AND <integer> event_name)

The and_list may be:

- a single event name
- two or more ANDed single event names.

The "&" character may be used instead of keyword AND, and the "|" character instead of OR.

A special AND expression must be enclosed in parenthesis, if it is ORed with another such expression. If it is used alone, parenthesis may be omitted.

In other words, any "special element" (WHERE, ALL, grouping integer, optional event) may appear only inside an ANDed (and bracketed) subexpression, which may only be ORed to the other parts of the triggering condition expression. Thus the special elements may appear only inside one level of brackets (which are mandatory when there are other OR parts), and there may be no ORs inside these brackets.

NOT operator is not used in GRAPES-BM.

If and_list in a special_AND_expression is followed by yet another event (connected via AND ALL or AND<integer> options), then the name of this additional event may not appear in the and_list. For example, e1 AND <2>e1 is invalid, use <3>e1 instead.

The general form for triggering conditions can be summed up as follows:

triggering_condition::=	and_term {OR and_term}*
and_term::=	stand_and_term | (special_and_term)
stand_and_term::=	stand_factor {AND stand_factor}*
stand_factor::=	event_name| (stand_expr)

stand_expr::=	stand_and_term {OR stand_and_term}*
special_and_term::=	and_list WHERE bool_expr |
	and_list AND ALL event_name |
	and_list AND ALL event_name WHERE bool_expr |
	and_list AND <integer_const> event_name |
	<integer_const>event_name |
	and_list {[AND event_name]}*[AND and_list]
and_list::=	event_name {AND event_name}*

Examples of valid triggering conditions

e1 AND e2

e1 & e2

e1 OR e2

e1 | e2

e1 AND (e2 OR e3 OR e4) AND e5

e1 AND e2 OR e1 AND e3

e1 AND ALL e2

<3>e1

e1 AND <3>e3

e1 AND e2 WHERE e1.f1=e2.f1

(e1 AND ALL e2) OR (e1 AND <3>e3) OR e5

(e1 AND e2 WHERE e1.f1=e2.f1) OR (e1 AND ALL e3) OR (e1 AND (e4 OR e5))

(e1 AND ALL e2)

e1 AND e2 [AND e3]

5.4.3 Semantics of triggering condition

If standard Boolean expressions are used, they have their intuitive meaning. Any minimal group of events, which satisfy this Boolean expression is used for triggering. Subexpression e AND e also requires only one instance of e to be present, which may have arrived from any of the sources (and only one instance is consumed). Similarly, e OR e is the same as e. If two instances of an event are necessary for a task to trigger, notation <2>e must be used.

The special AND expressions are used for group or selective triggering. Thus

event1 AND ALL event2

is triggered, when there is one event1 and one or more event2 present in queues of the task. Then all instances of event2 present in the queue are consumed together with one instance of event1.

Integer qualifier (which must be a constant) is used to define fixed size "packaging" of events, e.g.,

e1 AND <5> e2

requires one e1 and five e2's to be present in queue, and they are all consumed at triggering.

If the and_list contains more than one element, just one instance of each is taken together with the required number of instances of the last specified event. ALL and integer grouping may be applied also to timers (in the role of the last event), but this is not a frequently used construction.

The expression <5>e2 alone is also considered to be a special AND expression. It is used to specify just a package of events e2 as the triggering condition. <1>e1 is considered to be identical to simply e1, and, consequently not a group triggering expression.

The WHERE condition is a Boolean expression operating on incoming event fields and task attributes. Only the event (or events) from the and_list (beside which the WHERE condition appears) may be referenced. The condition itself is a normal Boolean expression containing relational operators on arguments, ANDs and ORs. The semantics is that only the event (or event groups) satisfying the condition is used in triggering. Non-matching event instances remain in queue. For example,

ev1 AND ev2 WHERE ev1.x1=ev2.x2

says that only those event pairs of ev1 and ev2, where the corresponding fields match, are taken from queues and consumed for triggering. Elementary fields from any level (using the appropriate qualification) may be referenced in WHERE condition.

Any task attributes may be referenced in the condition, and their values are specially computed at that moment. If the value happens to be undefined (NULL), any comparison of it to any event field returns the answer false.

Warning. If one of the attributes used in WHERE is based upon a random function, the attributes value used by WHERE may differ from the "actual" value (used in task statistics).

When ALL is combined with WHERE, only those instances of ALL-event which satisfy the WHERE condition, together with appropriate singular event instances for and_list make a "package" of events, which triggers a task instance and is consumed from the queues. Example:

e1 AND e3 AND ALL ev2 WHERE e1.x=e3.x AND e1.x=ev2.x

Here at least one matching ev2 must be in the package. All non-matching ev2 remain in the queue.

A special notation like

e1 [AND e2]

where the AND-symbol together the event name following it is enclosed in square brackets, may be used as the and_list forming the special AND expression. This notation means that the bracketed event is not mandatory for triggering, but when an instance of this event is present, it does take part in the triggering (and therefore is removed from the queue). Remember that if an event arrow enters a task symbol but the event is not present in the triggering condition, then instances of this event simply remain laying in the queue (except in the case where the triggering condition is completely absent). Square brackets may not be combined with the other special expression facilities (WHERE etc.). There may be more than one bracketed event, e.g.

e1[AND e2][AND e3]

If several of the ORed AND expressions are true simultaneously, the first of them (from left to right) is actually used for triggering. More precisely, any standard Boolean expression which is a top-level OR-part of the triggering condition, is internally converted to its disjunctive normal form. As far as possible, the

order of events in the original source expression is retained. Thus after this transformation any complicated triggering condition is an OR-expression, where each AND-term is either a simple ANDed list of events, or a special AND-expression (i.e., one containing ALL, WHERE, etc.). If a triggering condition is already in the disjunctive normal form, it is not transformed. During execution, each AND-term (from left to right) is checked, to test whether all events in this term are present (at least one instance). For special AND-expressions the additional requirements also are checked. The first term thus found to be true, is used as the actual triggering set.

5.4.4 Control flows in triggering and semantics for occurrences

Nameless events represent pure control flow. They never appear explicitly in the triggering condition. They either all are ANDed to the explicit triggering condition (if the triggering condition is a simple AND or any more complex one), or all are ORed (if the triggering condition is a simple OR).

If more than one control flow enters a task, a separate queue is assigned to each of them. More precisely, a potential input queue is assigned to each referenced task name associated to an incoming control flow in a TSD (regardless whether they appear in one referenced task symbol as a name list or in several).

But there is a general convention, that in every occurrence of a task only those input queues are built which potentially may receive an event (or control flow) in this occurrence according to connection rules (see 6.5). The simple AND requires that an instance of event from each existing queue must be present (and is consumed).

To sum up, this rule implies the most natural semantics, that in each occurrence of several ANDed (by default) control flows only all those really entering this occurrence are required for the task to start.

In OR cases, the presence of one of the control flows is sufficient for triggering.

The other consequence of this convention on queues and simple AND triggering, is that one occurrence of such a task may have, e.g., e1, e2, e3 as incoming events and the other only e1, e2 and both will be normally triggered (TSD will have e1, e2, e3 as incoming events and up to five referenced task symbols in this case).

Complex events may not be used for direct triggering of tasks, they must be refined first. They also may not be used for implicit triggering of elementary tasks (i.e. they are not allowed to enter an elementary task).

5.5 PERFORMER expression

The PERFORMER expression (see PERFORMER section in Fig. 5-2) sets the criteria by which a performer or group of performers is to be selected from the ORG diagram to execute the given task. These performers must be available before the task can start. Performers are taken from the ORG diagram of the business model.

5.5.1 Syntax of the performer expression

Any element of the ORG diagram (unit, position, resource) may be referenced as a performer in the performer expression. In the simplest case, the performer expression is simply one of the available performer names, e.g. *secretary*.

The general form of the performer expression is a Boolean expression built from performer_elements using AND, OR operators and parenthesis. The "&" and "|" characters may be used instead of keywords as well.

In the simplest case the performer_element is a organizational unit name, position name or resource name from ORG diagram. If a name is not unique diagram-wide, then it should be qualified by including corresponding higher level names from the ORG tree, e.g.,

SE_laboratory.Programmer

SE_laboratory.Computer.

If the performer is a multiple object (multiple unit, multiple position, multiple resource), the number of performers (resources)actually necessary may be specified (before the qualified name), e.g.,

<3> SE_laboratory.Programmer

<3> SE_laboratory.Computer

If no number is specified, one instance from the multiple performers is assumed.

The number of performers may be used also if there are several equally named position elements at the same level (from this point of view it is the same as if there were one multiple position with the appropriate number).

The performer expression will often also contain the specification of a necessary competence list (after the keyword WITH), e.g.,

<3> SE_laboratory.Programmer WITH COMPETENCE = Pascal, Cplus

AND-semantics are assumed for the competence list, i.e. both competencies are required here simultaneously.

If a performer or a resource is occupied by the given task only partly, e.g., only at 70% level, then it is specified as follows

<3>SE_laboratory.Programmer WITH COMPETENCE=Pascal FOR 70%

The keyword ANY may also be used instead of position in a performer expression, e.g.,

SE_laboratory.ANY WITH COMPETENCE=Pascal

Some more examples of performer expressions follow:

(<2>SE_programmer AND SE_computer) OR(<2>Artif_int_lab.Programmer
WITH COMPETENCE=OPS5)

(perf1 AND res1) OR (perf2 AND res2).

The syntax of the performer expression is the following:

performer_expression::=perf_and_term {OR perf_and_term}*

perf_and_term::=perf_factor {AND perf_factor}*

perf_factor::=performer_element | (performer_expression)

The general syntax of performer_element then is the following:

performer_element::=unit_perf | posit_perf | res_perf

unit_perf::= [num] {unit_name.}* unit_name[comp]

posit_perf::=[num] { unit_name.}* posit[.employee_name][comp][percent]

res_perf::=[num] { unit_name.}* [posit_name.]{resource_name.}* resource_name[comp][percent]

comp::=WITH COMPETENCE = competence_list

posit::= position_name | ANY

percent::= FOR integer_constant %

num::= <integer_constant>

competence_list::=competence_name{,competence_name}*

In general, the referenced names should be in accordance with the ORG diagram. Unique names at any level may be unqualified. Non-unique names must have necessary qualifications (unit_names, composite resource names) which make them unique.

However, it is permitted also to use "incomplete" specifications, i.e., when there are equally named performers in several places of ORG diagram, then by omitting some of the highest level qualifications, we can have access to all these places, e.g., to programmers from several departments. It is not permitted to omit "middle" qualifiers, each element must match to a tree fragment.

If the position is qualified by employee name, only the specified one is seized. This facility makes sense, if there are several similar positions distinguished only by employee names.

ANY position may be the lowest item in a performer element or may be followed only by WITH COMPETENCE specified. No resource or employee name may follow ANY. On the other hand, ANY may be preceded by unit specification, or used alone. When used with a unit specification, it means any position directly under this unit, while when used as a single keyword it means any position in the entire ORG diagram.

5.5.2 Semantics of the performer expression

When a compound organizational unit is specified as a performer, this includes all positions and resources from the specified unit in the performer expression. When the unit is seized none of the components of the unit are available for another task. Similarly, a composite resource means all its components. Position and elementary resources mean just the specified objects.

x AND x is the same as <2>x. Therefore *dept1* AND *dept1.secretary* makes no sense (the first element already requires the whole *dept1*). The number of required performers should not exceed the number of available ones. If no required number is specified, 1 is implied.

The FOR option does not affect the seizure of a performer or a resource - it is always seized for 100%. The cost is also not affected. The only effect of the FOR option is within performer statistics, where productive utilization is computed accordingly.

Performer availability periods are taken into account only when starting a task. If the task execution period runs over into an unavailability period for a performer, the performer completes the task in accordance with its duration.

In fact, the availability of specified performers acts as part of the triggering condition. If the triggering condition is true for some event group in a task's queues but none of the specified performer combinations is available, no triggering occurs, and the events remain in queue. Certainly, events with limited persistence, like timers, (see sec. 4) may vanish from queue while waiting for performers, so these event instances may trigger no task at all.

5.6 Other elements of task body

Now let us describe the other elements of task body. Fig. 5.2 shows an example of a complete body.

Triggering conditions and performer sections were already described above.

Section **TYPE** specifies the type name of the task. If no user defined type is used, the section is empty. The type specifies which attribute table is used for task attributes. Untyped tasks have no attributes, except the predefined ones.

INFORMAL DESCRIPTION, OBJECTIVES and **CONSTRAINTS** sections contain any informal text.

EXECUTION MODE section may contain one of the keywords: MANUAL, AUTOMATED , INTERACTIVE.

PRIORITY section has the syntax

PRIORITY : integer_const

with zero as the default value (the highest priority = 0, so the greater the constant, the lower is the priority). Priority governs the competitions of tasks for performers. Explicit priority greater than zero must be defined for "background" tasks, thus allowing normal tasks to seize performers as first.

The precise semantics of priority.

Let us assume that several tasks are ready to trigger (i.e., there is at least one triggering event set in each task's queues) but they are not triggered because no performer (common to all of them) is available. When a required performer becomes available, then among tasks which could now be started, the one with highest priority is selected to start.

If tasks compete for different performers, their relative priority has no effect on their starting order (i.e., all tasks are started as soon as possible). If there are ORed performers, for each performer set becoming available there is an independent competition.

A task being executed is never interrupted by a higher priority task (non-preemptive scheduling).

There are two **predefined attributes**, DURATION (of type duration), COST (of type float) for each task.

The DURATION attribute is described in the DURATION section of the task description. **COST** has no explicit description in TSD. Instead, the value of this attribute is computed dynamically, using DURATION from tasks being performed and the COST PER HOUR attributes of the performer(s) selected to perform this task from the ORG chart (namely, the actual duration is transformed to hour units and the obtained float value is multiplied by the appropriate cost per hour value). The COST attribute may be referenced in other formulas, however.

Cost per hour is summed for all performers used. Efficiency (which affects the duration) is also implicitly taken into account.

If a compound unit is defined as a performer (but not the elements of it), then the COST PER HOUR for the whole unit is used (if it is present), otherwise the sum of the costs of a unit's direct constituents is used.

The **DURATION** section may contain a proper duration constant, a named constant from SP, a random duration function, or a duration type expression, containing as arguments the above mentioned values, and in addition, attributes of incoming events and task attributes.

Restriction : only attributes of events which are always present may be referenced, an execution error message appears when missing event is referenced.

If a group triggered event is referenced, then the first instance of it is taken. Any task attribute may be used in a duration expression, and the derived values are specially computed at that moment. If the expression results in NULL value, zero duration is assumed.

Warning. If an attribute with a random value is involved, the value may be different from the final value of the attribute.

Examples:

"3h"

"2d:10h"

EXPONENTIAL("2h")*order.quantity

line.duration*letter.length

ATTRIBUTES section may contain some of the attributes for the given task type. The attributes present in the corresponding ATR but not included in ATTRIBUTES section retain their definitions (default value or formula, with formula having priority), if such are provided in ATR's. Those without definition have undefined (NULL) value. No assignments are permitted to string attributes in this section, if the model is to be used for simulation.

The presence of attributes in the ATTRIBUTES section completely redefines their value by the provided expression (which may be a constant or a proper formula). If a new formula is defined here, there are wider possibilities for its arguments. In that case input event fields may also be used as arguments. They are referenced as *event.field* (or *event.field1.field2.field3*, if the record is nested, the actually referenced value must always be elementary). It must be ensured that the event type has such a record field, and that the task is actually triggered by such an event (otherwise NULL value appears together with a warning at runtime). When an event has an elementary type, just the event name is used for referencing its value. Predefined task attributes may also be used in formulas. Each attribute setting is terminated by ";" character. After the last attribute, the ";" character may be optionally inserted. Thus attribute setting is a sort of assignment statement.

If an event e2 appears in a "group triggering form", i.e., e1 AND ALL e2 or e1 AND <10> e2, then SUM, MAX, MIN, AVG operations may be applied to fields of e2, e.g., attr5: SUM(e2.x1). If an ordinary arithmetic operator is applied to such a group event, the first instance is taken.

Random values may be used freely in attribute formulas. See more on expressions in section 11.

Each possible attribute from the corresponding ATR table may be redefined only once in the ATTRIBUTES section. The order in which the attributes are redefined in the section has no semantic meaning.

The retained attribute definitions from the ATR table and the redefined ones from the attributes section in tasks together are sorted in an order where an attribute referencing another attributes in its final formula is evaluated after the referenced ones. If a circular reference is found, an error message is generated (i.e., such an ordering is forbidden). The attributes in a TSD are evaluated during simulation in the order defined by this sorting.

A special case is attributes of transactions (non-elementary) tasks, which are evaluated at the corresponding transaction end (see sec. 7). Besides other attributes of the task, formulas in transactions may contain also attributes of other (elementary) tasks inside vertical operations (SUM, MAX, MIN, AVG). Here "vertically processed" attributes are referenced purely by their names. Any arithmetic or duration type attribute from any ATR table formally may be referenced in a vertical operation. If several ATR tables contain equally named attributes, their types must also be equal, if these attributes are being "vertically processed" in transaction attribute formulas. Formulas of transactions may not contain event fields. Any task instance having the referenced attribute and which belongs to the transaction instance is taken into account. See more on it in sec. 7.4.

MAX INSTANCES section defines the maximal allowed number of simultaneous instances of the task. This is an additional absolute limit on the number of instances, besides the performer selection expression together with the ORG diagram which also define a limit on the instance number.

ALTERNATIVES section appears only when there are several alternative refinement TCDs under a complex task. It contains their names and probabilities. The GRADE tool supports automatic extension of the alternative section when new alternatives are inserted directly in the model tree. For top-level tasks, the Alternatives section is valid, when this top level task is "called" in some TCD (see 8.2).

In general, the *Alternatives* section effects only the routing into the given task (see 6.5). If alternative TCDs (including those at the top level) have autonomous activities inside (e.g. timer), they all function in parallel, irrespective of probabilities.

Percents may be absent from one or all alternatives in the section, and then 100/n is assumed for each. It is not permissible to specify percentages for some and not for others.

Type, Attributes and Alternatives are the only sections of the task body which are operative on complex tasks. All other sections are actually redefined by their refinements.

5.7 Decisions

Decisions can have detailing which are statements placed inside the decision symbol. The complete syntax of decisions is as follows:

decision_name

[*formula*]

[*probability*]

where

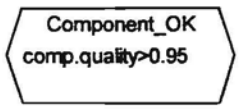
formula is *Boolean expression* | ELSE | ALWAYS

and *probability* is [number%] [EXCLUSIVE]

Formula and probability are optional. The formula may contain attributes (including derived ones) and input fields (such as attribute formulas). Typically either a formula or probability is defined forall of a task's decisions , but the options may also be mixed. Decision names must be unique within one task, and names must always be present.

The formula is any Boolean expression (see section 10) containing attributes and input event fields. It should be reminded, that for "group-triggering" events the decision is taken once for the whole group. Therefore only vertical operations on such event fields should be used. If a group event is referenced without a vertical operation, the first instance is taken.

An example of decision:



The decision formula may contain also special built-in function **Is_triggered_by** (event_name). The function is true, if at least one event with the specified name was actually consumed during the triggering of the current task instance. The function makes sense, only if there is a usage of OR in the triggering condition (otherwise the function has a constant value).

In V.3.0, non-exclusive decision semantics are assumed, where several branches may be activated simultaneously. This assumption is more general than the previous "exclusive" one used in version 2.0..

First, let us explain the new semantics for formulas. Each decision formula may be true or false independently of others, and if the formula is true, the branch becomes active (i.e., the associated outputs are sent). Two specific "formulas" defined by keywords ALWAYS and ELSE also may be used. ALWAYS is just a syntactic equivalent for constantly true formulas, ELSE becomes true, if no other decision branch is used. The standard exclusive style may be obtained, if formulas are mutually exclusive.

Now, let us consider the probability based decisions. There will be two syntactic possibilities for the probability specification

n%

or

n% EXCLUSIVE

where n is a non-negative integer or real constant, not exceeding 100.

If there is no EXCLUSIVE option for any of the decisions, then each of the decisions becomes active irrespective of others, with its specified probability, e.g., a decision with 30% value becomes active with probability 0.3. Decision with 100% value becomes active always.

If on the contrary, all decisions have EXCLUSIVE option specified, then the sum of percents should be equal to 100 (if all EXCLUSIVE branches have the percent specified). If the sum exceeds 100%, a warning is issued during analysis (and a branch may become unreachable during simulation). Only one of the decisions may become active in an EXCLUSIVE case, according to the percentage specified. If the sum is

just 100%, just one decision always becomes active. If the sum is less than 100%, then no decision becomes active with the probability (100 - sum)/100 (no analysis warning appears in this case).

These two cases are the normal ones for probability based decisions. However, EXCLUSIVE and non-exclusive decisions may be freely mixed. In that case EXCLUSIVE decisions (which again must have a sum not exceeding 100) form a group, which behaves independently of the other (non-exclusive) decisions and activates zero or one decision. On the other hand, non-exclusive decisions also function independently of exclusive ones, i.e., each decision is independently activated with the specified probability.

ELSE decision may also be combined with probability decisions, with natural semantics (it is used if none of the probability decisions is selected).

Only one ELSE decision is permitted per task - both in the formula and probability case.

Yet another type of probability is possible by specifying simply the keyword

EXCLUSIVE,

without any percent specification. This option is used simply to specify the exclusive OR relation between decisions (only one is possible). From the formal execution point of view, 100% (or less, if there are some EXCLUSIVE decisions with percentage specified) is divided equally among them.

The EXCLUSIVE keyword is provided for better comprehension, since probability without a percent and without EXCLUSIVE is the same as if nothing would be specified at all (and EXCLUSIVE with equal chances is assumed in that case). For really non-exclusive decisions use n% case.

If nothing is specified for any of task's decisions (i.e. neither formula nor probability is selected), a probability of (100/n)% (exclusive) is assumed for each. But then nothing must be specified in any of a task's decisions. It is not allowed to mix specified and unspecified decisions for one task. All diagnostic messages on decision inconsistency inside a task are at warning level.

5.8 Output events

Output events can also have details, which are used if the data values carried by message events are significant in the model.

The details may contain the SET option for setting values of output message fields and REPEAT for increasing the quantity of outgoing events. The syntax for SET is

```
SET   field1=expression;
      field2=expression;...
```

Expression may contain task attributes and input event fields (as in the ATTRIBUTES section). The same syntax for field referencing is used. The expression type must match the field type. See more on expressions in sec. 10. Each field setting is terminated by a semicolon. After the last (or sole) field setting the semicolon is optional.

The REPEAT option has the following syntax:

REPEAT integer_expression

This option may be used to send several messages (with equal data) upon task completion, e.g.

SET field1=x+1;field2=event1.a; REPEAT event1.b.

In the case of events with elementary types, the form

SET VALUE=expr

is used. If the event has a nested record type, qualified field names are used:

field1.field11.field111=expression;

Only elementary fields may be on the left-hand-side of such an "assignment", i.e. no record assignment is permitted in this version.

The repeat option may also be used alone,

REPEAT integer_expression.

SET and REPEAT options appear as text below the event name.

Remark. REPEAT may not be used as a record field name when the event has this record data type.

There is special convention on message passing through the task. If there is an incoming event and outgoing event of the same name, the field values of the incoming event are passed without changes to those of the outgoing event, without any explicit SET option for it. If there is a SET option for such an event, only the event fields set explicitly in the option have the new values, the other go unchanged.

For more complicated cases one more convention is assumed. If names of input and output events are different, but they have the same data type (i.e., they reference the same type name in ET), then a similar field value passing from input to output occurs. In the case that several input events with the same record type together have triggered the task instance and the output event also has the same type, one of these input events is taken for value passing.

Another special feature is multiple event passing, when the corresponding incoming event is "group-triggered" i.e., in AND ALL or AND <n> connection, and there is an outgoing event with the same name. In this case all instances of the incoming event are passed through the task. SET option (if any) should reference only task attributes (or attributes of "single" events) in that case (i.e., the updated fields are computed only once and are always the same for all instances). Other outgoing events, as always, are generated in only one instance. Their field values should depend on a group-triggered event only via vertical operations. If a group triggered event is referenced without vertical operation, the first instance of it is taken.

If REPEAT is specified for multiple events, each instance is copied the specified number of times.

SET and REPEAT may be used only for named events. Outgoing control flow always have only one instance. Complex event may never be sent by elementary tasks.

If there is no SET option for an event and none of the default value transfer rules apply, the field values of the output even are undefined (NULL).

5.9 Input events

There is also one possibility provided for detailing **input** event, and is really used only for simulation. Namely, for input events a spontaneous generation option is provided in the form

TIME (time_specification)

or

REPETITION(duration_expression)

may be used. Time_specification has the same syntax as for timer definition in ET (see sec. 4.2). The same restrictions as in ET apply for the duration expression (except that arithmetic expressions may not be used here). The time moments for spontaneous insertion of events in a task's input queue (in a TCD) is defined as for timers.

This option is used to define system load generators "on the fly", i.e. when timer-like behavior is necessary, but an explicit timer for some reasons is undesirable. The feature may be used only for named events (not control flows).

There are two preconditions for the spontaneous event generation to function. First, it functions only, if the task is elementary. The typical usage of the feature is when we want an external task to generate events to be processed by the system, without using an explicit timer symbol. For complex tasks it is simply ignored. Second, there must be an incoming arrow with the given event name in the TCD (more precisely, there must be an incoming route for this event from some task, see 6.5). Certainly, for the generator to function properly, the other end of this arrow should start from a "dead" task - an external task without any stimulus (or external without name) as a rule. However, it is not an error, if the other end starts from a "live" task, then the two event flows will mix together.

Details of input event are also shown in TSD as a text below the event name.

5.10 External tasks

External tasks also have TSD diagrams in version 3.0, which look the same way as those for internal tasks. This means that external tasks also appear in the model tree, and they can also be refined via TCD. They may also have a type.

This means that there is no more formal syntactic or semantic difference between internal and external tasks. Externality has no impact on simulation semantics definition or statistics. Internal and external tasks are distinguished at the informal level, to improve model readability.

The sole special feature of external tasks, is that it is allowed to have unnamed external tasks in TCDs. Such external tasks do not appear in the model tree and have no TSD. The use of such tasks is for modeling only. From the simulation point of view they are considered as "dead" tasks. Events which would be sent to them are simply discarded since they have no input queues. They never generate any events. However, these tasks are considered as existing from the routing point of view (see 6.5). Thus, if a route comes from such an external unnamed task, the queue is built at the other end of such route (to allow an appropriate spontaneous generator to make this route "live"). Unnamed external tasks in TCD may induce also external referenced tasks without names in the refinements of their neighbors (in the TSD and the refinement TCD), which are used only for routing.

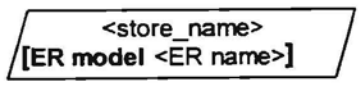
Remember that some external tasks normally are workload generators of a system. These tasks should be triggered spontaneously. This is described usually with a timer (using also random values, as a rule) being the only triggering event of such a task.

It is recommended to specify external performers for external tasks, but formally there is no links between these two kinds of externality.

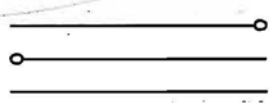
5.11 Data stores and data objects

Data stores and data objects have only informal semantics in GRAPES-BM version 3.0.

Each data store has a name, and potentially, ER description:

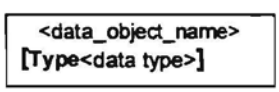


ER_name must be the name of a visible ER model. If the ER_name is omitted, name equal to data store name is assumed. But there may be no ER-specification at all since data store may be completely informal (or contain even physical objects). The only reference to entity names is in AT for this task, there entities from the ER model corresponding to the specified data store must be used. The database name in AT should coincide with one of the data store names present in the TSD. For informal data stores, AT is not used and AT remains blank if only informal data stores are present in a TSD. Access paths are of three types (read, write, both) and may have optional names. These names are completely informal. No consistency between access paths and the AT of a TSD is checked. The graphical form for access paths is the following:



Data stores are meant to represent persistent data existing in a business system. Typically such objects are data bases, but also all kinds of archives belong to this category. Persistent stores of physical objects (stores in warehouses etc.) should also be represented this way.

Data object symbol is the following



Data object name may be any, type, if specified, must be defined in a visible DD. Any type may be referenced. Optionally named access paths to data object have the same form as for data stores. No links to AT at all are used for data objects.

The informal use of data objects is a temporal data object created by one task and used by others, it normally persists during one transaction. Sometimes this feature is used as a substitute for event sending between two tasks, when a common data object is more natural. From the programmer's point of view data objects should be understood as global variables.

Both data stores and data objects are ignored in simulation.

6 Task Communication Diagram

6.1 Role of TCD diagrams

Task Communication Diagrams (TCD) are the main facility of GRAPES-BM for describing business system behavior. They are used to refine large tasks as chains of smaller tasks linked via events. TCD diagrams show how events generated by one task are passed to another one to trigger it in turn. Timers are also represented in TCD diagrams.

Business system refinement is started from **primary** tasks for which the highest level TCD diagrams are built. Tasks appearing in such a TCD diagram (i.e. their TCD diagrams) normally are placed in the model tree directly subordinated to the corresponding top TSD diagram. Some of these next level tasks may have their refinement TCD diagrams in turn, until the desired detailing of business system behavior is described. In lower level TCD diagrams the event linkage between adjacent TCD levels is also shown. This is done by **referenced task symbols** (and referenced timers) which appeared already in TSD diagrams. Referenced task symbols are the successors to the remote task symbols used in GRAPES-BM version 2.0, however, the precise syntax and semantics is not always identical. Fig. 6.1 shows an example of TCD diagram.

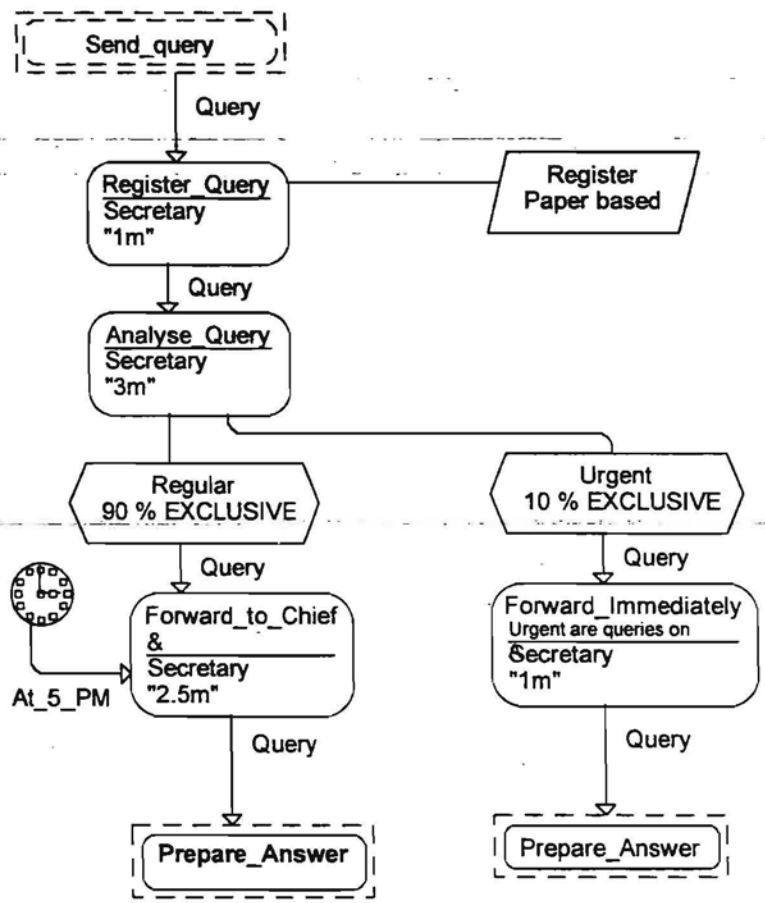


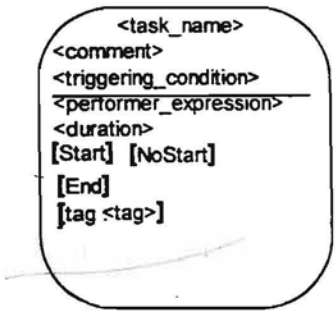
Fig. 6.1 Example of TCD diagram

6.2 Elements of TCD diagrams

Task and decision symbols and event arrows in TCD diagrams, besides their names, may have several textual sections, identical to those present in the corresponding TSD diagrams. In general, these sections must either coincide with the corresponding sections of TSD diagrams or be empty. In case of discrepancies, the formal information for simulation is taken from the corresponding TSD section. There are also new textual elements both for tasks and arrows, which can appear only in TCD diagrams. For each of the elements the role of each text section will be explained separately.

Any task symbol in TCD may contain also a WMF format picture.

6.2.1 Internal task symbol



The internal task symbol is the main element of task refinement in TCD diagram. The only mandatory textual element is its name. The name links the task symbol in a TCD to the formal definition of the task in its TSD. This definition is found according to visibility rules (see 1.2). The appearance of a task symbol in a TCD diagram is called a task **occurrence**. There may be more than one occurrence of the same task in one TCD (and in several TCDs also).

Triggering condition, performer expression and **duration** are copies of the corresponding sections of task definition TSD. Alternatively, they may remain empty even when these sections are present in the TSD. No extraneous information should be added. Special care should be taken in case of several occurrences of the same task. It makes no sense, e.g. to specify performer p1 in one occurrence and performer p2 in the other, since both must be equal to the performer specified in the performer selection expression in the TSD diagram. In the event that there is a discrepancy between TCD and TSD, it should be reminded, that formal information for simulation is taken from TSD. As far as possible, GRADE editors try to ensure consistency, by automatically transferring nonempty textual sections from TCD to TSD.

The formal syntax of triggering conditions, performer expressions and durations in TCDs is literally the same as in TSDs and is to be found in sections 5.4, 5.5, 5.6 respectively.

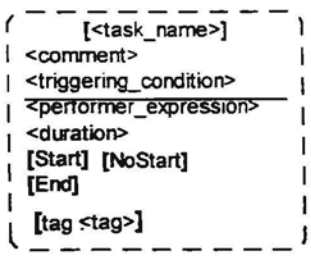
Comment is an arbitrary comment for a task occurrence (and may be different for several occurrences). It is not copied to a task's TSD. There a comment may be part of a task's informal description.

Occurrence tag is a formal identifier used to distinguish several occurrences of the same task in one TCD. Its sole use is identification when viewing simulation results (directly in the simulator or via the Trace Browser) and for defining show-boxes (sec. 6.2.11).

Start, Nostart and **End** options are used for explicit transaction control related to the task occurrence (see more in sec. 7.3). They also never appear in a TSD.

Any task in a TCD is either a transformation or a decision task. If it is a decision task, the task symbol is connected to its decision symbols via simple lines.

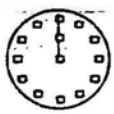
6.2.2 External task symbol



External task symbol has the same formal properties as the internal task symbol and is refined in the same way with a TSD in the model. The difference between external and internal task is completely informal, just to emphasize that some activity is performed outside the framework of the business system under consideration.

There is only one additional feature for external tasks. External task may be unnamed. Then it has no defining TSD. From the formal execution point of view, it is called a "dead task". It sends no events, events to be sent to such task are simply discarded (i.e. not sent at all).

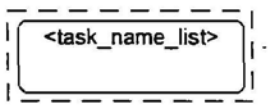
6.2.3 Timer symbol



Timer symbol defines an independent timer occurrence (determined by the timer event name leaving it). This timer occurrence, determined by its definition in the ET, sends the corresponding timer events.

Caution. Do not confuse this symbol with a referenced timer, which has no autonomous activity.

6.2.4 Referenced internal task symbol



where <task_name_list> ::= task_name{,task_name}*

A referenced task symbol in a TCD diagram represents one or more neighbor tasks in a TCD one level above the current one, to or from which the given event has been sent or received respectively, by the task whose refinement is the given TCD. The name or name list in the referenced tasks are equal to the mentioned neighbor name (or names).

More formally, referenced task symbols in a TCD must coincide with (or be subset of) referenced task symbols present in the TSD diagram whose refinement is the given TCD diagram. Events coming from these referenced tasks (or going to them) must be the same in TCD as in the TSD diagram.

According to their role, there are **incoming** and **outgoing** referenced task symbols in TCD diagram.

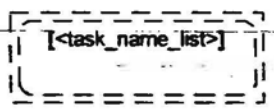
Incoming referenced tasks in TCD correspond to incoming referenced tasks in TSD diagram, and the same applies to outgoing ones. From incoming referenced tasks in a TCD, events go to internal (or external) tasks of this TCD, thus representing incoming links from the next upper level. And, respectively, events go from internal tasks to outgoing referenced tasks, thus representing outgoing links.

Actually, not the referenced task symbols themselves, but the pairs <referenced_name, event_name> must be the same in the TSD and its refinement TCD. It is permitted to redistribute name lists over several referenced task symbols associated with the same event name. Referenced task symbol (together with its associated event) may be duplicated in a TCD. Several event arrows may leave a referenced task symbol. See more on formal consistency rules in sec. 6.8.

It is forbidden to jump over levels. That means: it is forbidden to reference tasks other than internal/external or already referenced tasks from the next upper level TCD in referenced task symbols.

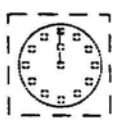
The special case is top level refinement of primary tasks. In the same way as in TSDs for primary tasks, it is permitted in such TCDs to reference another primary task in a referenced task symbol (if such a reference is already present in the TSD). See more on linking primary tasks in 8.1

6.2.5 Referenced external task symbol



Formally, this symbol has the same properties as the referenced internal task symbol. It is intended for use when the upper level neighbor (or neighbors) of a task are external tasks. Then, naturally, the corresponding referenced task symbol in TSD is also an external one. It is not formally considered an error, if externality one level above is ignored in referenced tasks of this TCD, or internal and external neighbor names are mixed in one symbol. By default, the editor distinguishes between internal and external tasks when building automatically the referenced task symbols (in TSDs and TCDs). The only syntactic feature is, that there may be a external referenced task symbol without any name. It corresponds to an unnamed external upper level neighbor.

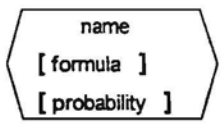
6.2.6 Referenced timer symbol



This symbol represents a timer one or more levels above the reference. It may be used if the TSD diagram whose refinement is the given TCD already has such a symbol. It must be associated with the same timer name as in the TSD.

It must be remembered that the referenced timer symbol has no spontaneous activity, unless the corresponding actual timer some level above it generates a timer event. The referenced timer then only helps to redirect the timer event to the required task in a lower TCD.

6.2.7 Decision symbol



The decision symbol is always connected to a decision task in a TCD. The only mandatory element in a decision symbol is the decision name, which must consistent in both the TCD and the task's TSD..

Formula and probability have the same syntax as in TSD (see 5.7):

formula is a Boolean expression,

probability is [number%][EXCLUSIVE]

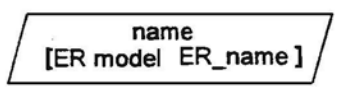
Only one of them may be present. If formula or probability is present, then it must have the same definition as in the corresponding TSD (the same as for task symbol sections). In the case of a discrepancy the formal value is always taken from the TSD. GRADE editors help to maintain the decision consistency between TCD and TSD, by transferring modified decision elements from TCD back to TSD. It is permitted to omit decision details in the TCD if they are in the TSD.

It is also permitted to have less decisions for a task in a TCD than in TSD (but not vice versa).

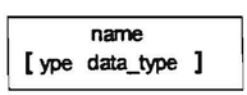
6.2.8 Data symbols

There are two of them:

Data store symbol



and **data object** symbol



Both symbols have informal semantics only and are not used in simulation. The syntax and intended semantics is the same as for these symbols in TSD (see 5.11).

It is recommended to maintain consistency between TSD and TCD for data symbols. GRADE editors try to help in this by automatically transferring data symbols from TCD to TSD. But there are no other consistency checks.

6.2.9 Event arrow

The event arrow is an arrow linking two task symbols in TCD diagram. It has the form:

[<event_name>][/<transfer_time>][/NOTID]

[set_option][repeat_option]

If the event arrow represents a named event or timer from the ET, the event name is mandatory. An unnamed event arrow represents a **control flow**. For control flows no other syntactic elements may be specified, i.e. the other elements are valid only if there is an event name.

If the transfer time is specified, it overrides the transfer time specified in ET. The same syntax is used for transfer time definition as in the ET except that arithmetic expressions may not be used here. Transfer time may be specified only for internal arrows, i.e., arrows connecting internal/external tasks (but not referenced ones). If, however, transfer time is added to an incoming or outgoing arrow it is simply ignored. If transfer time is not specified either in the ET or in the TCD, zero time is assumed. It is forbidden to specify a transfer time for timers.

NOTID option is used to prevent transaction TID transfer along with the event (see more on it in section 7.3).

Set_option and repeat_option are identical to those options in a TSD diagram at an output event (see 5.8). If there are additional details associated with some outgoing event in a TSD, the same information may be placed at the corresponding event arrow in a TCD. It is not allowed to place different data in the TCD, in formal processing only the data from TSDs is used. GRADE editors try to support the consistency by transferring output detailing from the TCD to TSDs.

It is permitted to enclose the event name on a path in square brackets, e.g.,

[ev1]

This means that the event may also not be sent by the issuing task. This notation is just an informal comment for modeling purposes. From the formal simulation semantics point of view the event is always sent. A real optional event sending, e.g., with a given probability must be specified explicitly by the decision for the issuing task.

Square brackets are not copied back from TCD to TSD (i.e., they only appear in the TCD). This notation may be combined also with additional texts on the arrow.

6.2.10 Access path

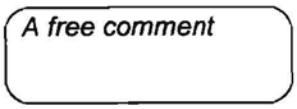
[<access_path_name>]

Access paths connects task symbols (internal or external) to data store or data object symbols. The path may have three forms depending on access type, in the same way as in TSDs. The name is optional. The element is completely informal in GRAPES-BM, and the semantics are the same as in TSDs (see 5.11). Data connections in TCDs should be consistent with those in TSD, but no formal checks are performed.

6.2.11 Auxiliary symbols

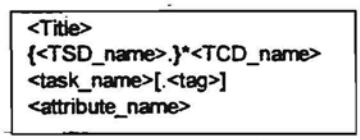
There are two such possible symbols in TCD diagram

1. **Free comment**



This symbol has no syntactic meaning and actually may be placed in other diagrams (TSD, ORG) too. Any text or WMF format picture may be placed there.

2. **Show box**



This symbol is not syntactically related to the TCD diagram in which it is placed. Its elements, however, must be valid

- qualified TCD name (i.e. TCD name with prefixed TSD names, starting from a primary task). Qualification may be dropped if TCD name is unique in the tree.
- task_name, followed by optional occurrence tag (tag is used if there is more than one occurrence)
- attribute name of this task.

The only use of such object is during animation of a business model, otherwise it has no effect.

6.2.12 Refinement of complex events

Another issue in TCDs is that of **complex** events. These events must have COMPLEX type already defined in ET (in the category column). Their component events must be defined in the corresponding column. When the actual event refinement is done in this refinement TCD, the following syntax is used on the arrow coming from (going to) a referenced task:

complex_event_name.event_name

The *event_name* is the name of one of the components, the *complex_event_name* is the name on the corresponding arrow in the TSD (and, consequently, on the arrow in the TCD one level up). The notation means that inside the current TCD only the *event_name* appears further (e.g. is used for triggering). The "complex qualification" appears only on input/output arrows in TCD, and it is not duplicated in TSDs (upper or lower). Any complex event must be refined before it is used in triggering. On the other hand,

elementary component events sent by a task at an appropriate level boundary must be "packed" into complex event. Refinement may not be used on horizontal arrows.

Linked primary tasks may also send each other complex events, then refinement may be done in both top level TCDs, or somewhere lower.

6.3 General rules of TCD structure

The TCD diagram is built from the elements in the previous section in a very natural way. However, some general rules on internal structure of TCD must be mentioned:

- event arrow may go

  - from any internal or external task to any like one (including itself). These arrows are called horizontal ones in the sequel. The number of arrows between two tasks is generally speaking unlimited.

  - from an incoming referenced task to any internal or external task. These arrows are called incoming events. One or more arrows may leave a referenced task

  - from internal or external task (or its decisions if it is a decision task) to outgoing referenced task (such arrows are called outgoing events). One or more arrows may enter a referenced task.

- it is permitted to use one referenced task symbol in the role of both incoming and outgoing referenced task.
- if there are several arrows between two tasks, all of them must have different event names. In particular, there may be only one control flow between two tasks.
- if the task is a decision task (i.e. it has at least one decision attached), all events may leave only decisions of the task, and not its body symbol.

6.4 Graphic layouts of the TCD diagram

From the language point of view the TCD diagram's contents is always the same irrespective of how it is displayed or printed. However, its visual appearance may be significantly altered by the user.

First, there are **long** and **short** forms. All examples shown so far were in the long form, when all textual items present in any of the diagram elements are also visible.

In the short form:

- in task symbols (internal and external) only name and comment remain visible
- in decisions only name remains visible

Texts on arrows is not affected.

All long form elements remain internally in place and regain visibility when switched back to long form.

Second, there are several graphic layouts available:

- vertical
- horizontal
- automatic
- manual

- tabular vertical
- tabular horizontal

In the first two layouts, all elements are automatically placed in fixed grid positions, so that the general event flow goes from

- from top to down, or
- from left to right respectively.

Certainly, any diagram may be transformed to this layout, with some arrows going in the opposite direction from the general event flow of the diagram. For simple "streamlined" diagrams, these layouts are the best, since the obtained ordering then corresponds to the real ordering of tasks in time.

In automatic layout, a compact allocation of elements is used, with the user having the possibility to select the place for a new element and the editor moving the existing elements in a minimal way to allocate space for the new one.

Automatic layout is suited for all kinds of diagrams.

Manual layout gives the user maximum control over the allocation of elements. Even the texts may be moved separately. But the user is responsible for the manual moving of existing elements when a new element is inserted. Manual layout should be used for presentation versions of diagrams and very compact allocations of large diagrams.

The two tabular layout modes with **lanes** (sometimes called **tabular layout** modes) are similar to the vertical or horizontal modes, respectively. The main difference is that separate lanes are allocated for each performer selection expression appearing in the diagram. The tasks containing the given performer expression automatically appear in the lane corresponding to the performer expression. The performer expressions themselves appear as the lane headings. For two tasks to appear in the same lane, the performer expression must be exactly the same. The layouts with lanes are well suited for diagrams with low variety of simple performer expressions. Then they show a nice table-like display of the tasks to be done by each performer.

There is the possibility of freely switching between all layout styles. The syntactic aspects of diagrams are remain unaffected.

Fig. 6.2 and 6.3 show the same example of Fig. 6.1 in vertical and horizontal layouts, respectively. Fig 6.3a shows an example of the vertical tabular layout, using an example with a number of performers.

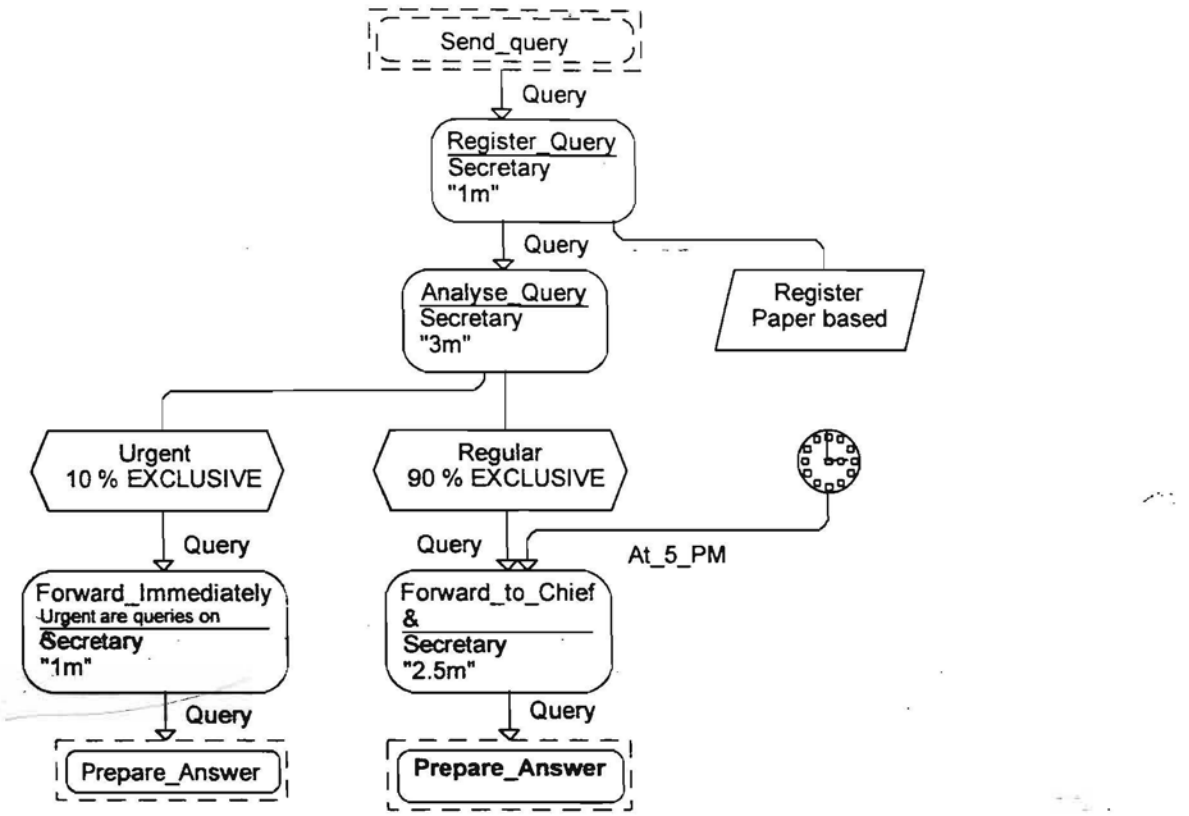


Fig. 6.2 Vertical layout

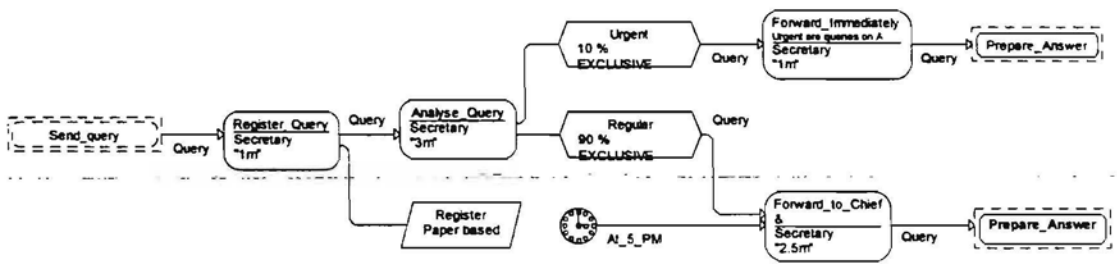


Fig. 6.3 Horizontal layout

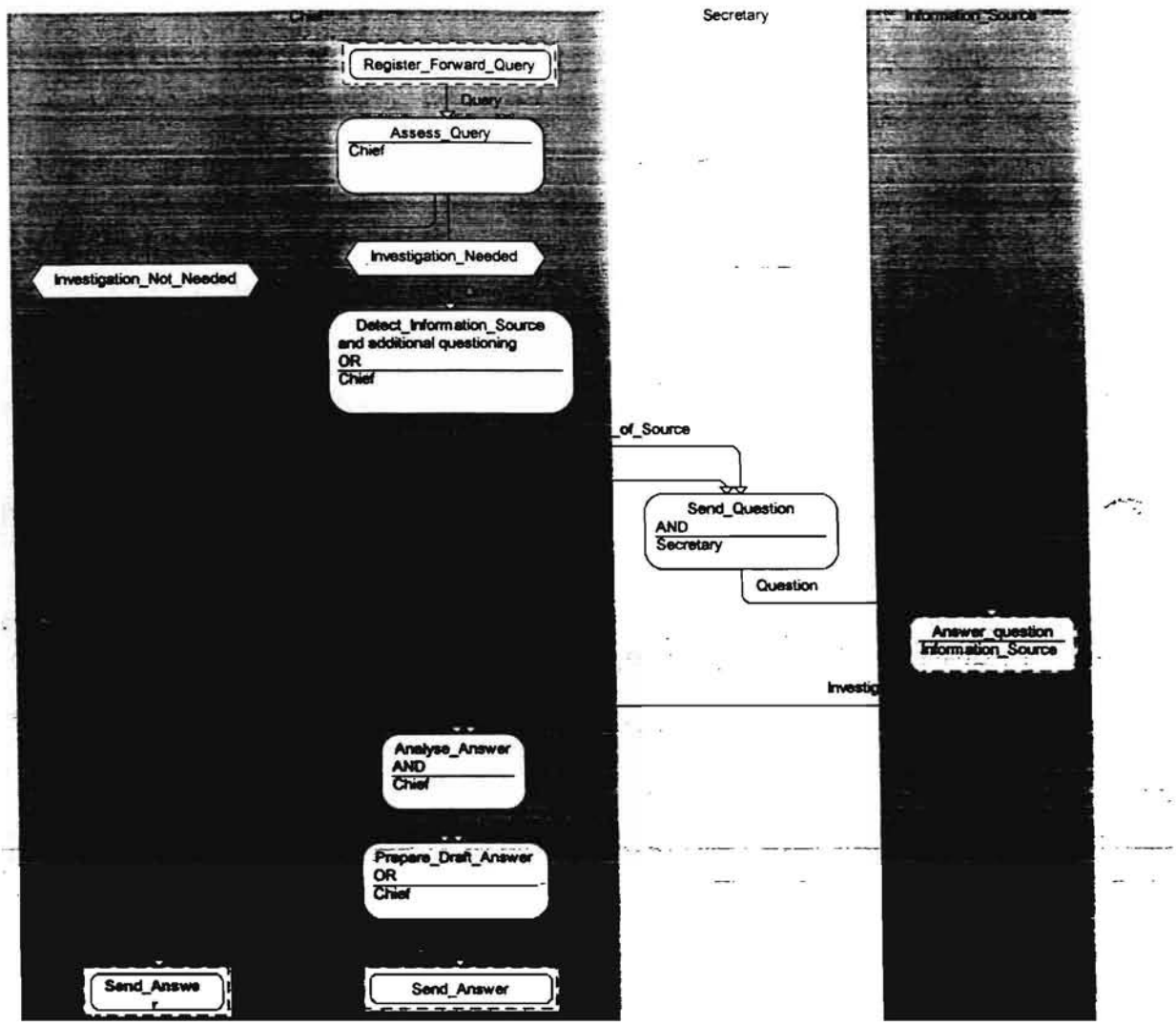


Fig. 6.3a Vertical layout with lanes

6.5 Links between TCD levels

Large systems can never be described by only one TCD level, a number of TCD refinement levels are used as a rule. Though in general, inter-level links are described by referenced tasks (which correspond to remote tasks in GRAPES-BM version 2.0), extended syntactic and semantic features are offered for describing those links. The main new feature is a new use of referenced task names.

In most simple cases, the inter-level links are defined in the most natural way where nearly all linkage elements are supplied by editors automatically. However, more sophisticated level structuring is also possible now, e.g., representing fragments of TCD diagrams like macros with many entries and exits. When a task has one occurrence, the refinement is very straightforward. Fig. 6.4 shows an example of simple refinement.

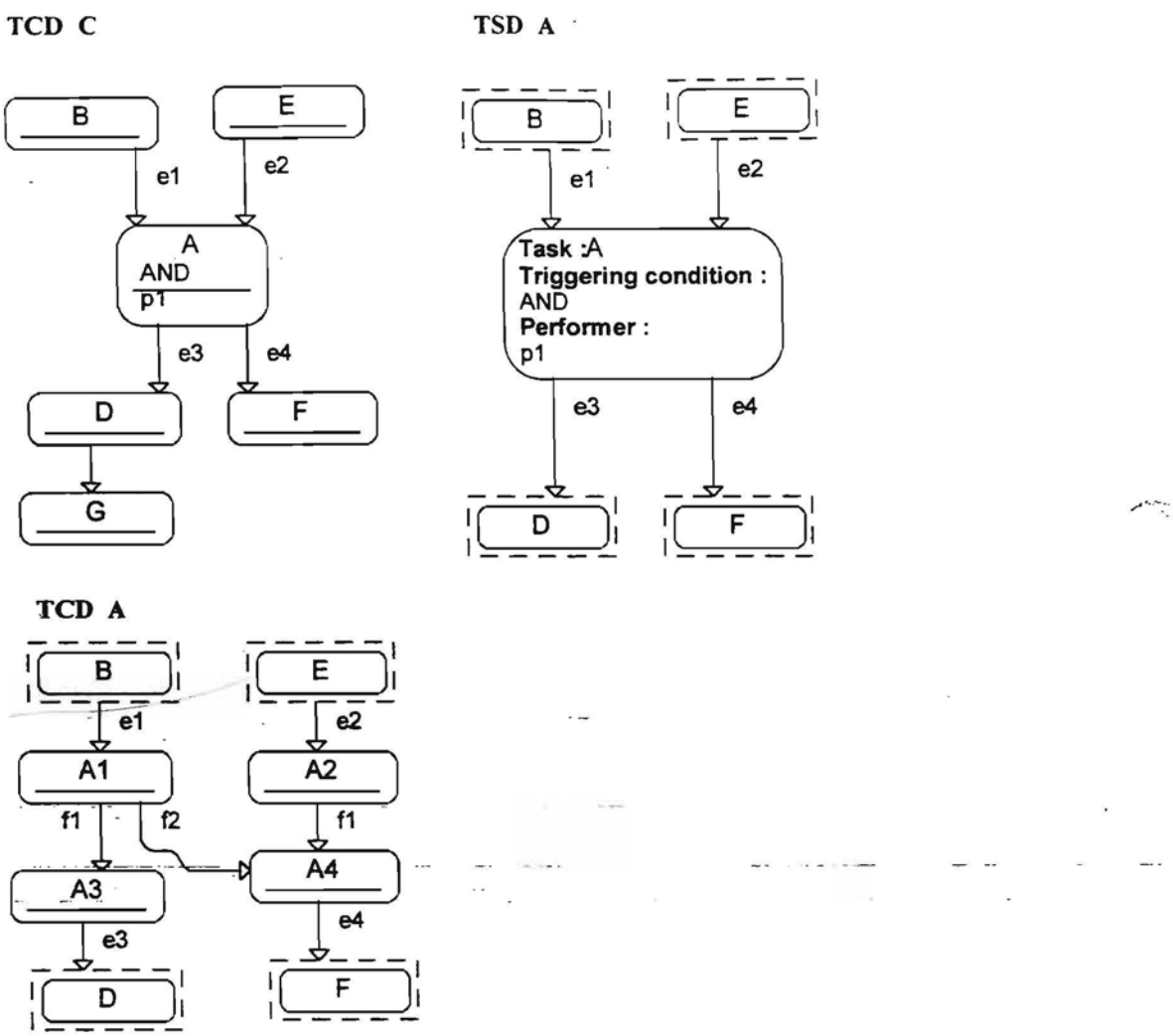


Fig. 6.4 Simple refinement

TSD for task A represents also its neighbors as referenced tasks, and they reappear in A refinement via TCD, showing clearly, e.g., that event e1 is routed from B to A1.

Often this event routing at level boundaries may be more complicated. A formal description of this routing follows.

Each event is sent by an occurrence of an elementary task. The event's destination is also one or more occurrences of elementary tasks, where the event is placed in the corresponding queues. In any event's route from its sending task to its destination queue there is just one event arrow linking the two tasks. If the sender and the receiver are in one TCD, and are linked directly by an arrow, the whole route consists of this arrow.

But often the event at first is routed via several outgoing referenced tasks, then it travels along the sole "internal" event arrow and then is routed via several incoming referenced tasks (see fig. 6.5). The internal arrow (the arrow from task A to task B in fig. 6.5) is called the horizontal link in what follows, the routing via outgoing referenced tasks - the upgoing link, and the routing via incoming referenced tasks - the downgoing link.

There is one special case when the explicit horizontal link is absent, namely the connection between two top level tasks (see more in sec. 8.1). Then the link is replaced by appropriate referenced tasks in two top level TSDs.

The precise event routing rule is the following. The outgoing arrow for the given event is analyzed. If it leads to an internal (external) task, the horizontal link is already reached. Otherwise, the referenced task name list is taken, and the following is repeated for each name in it (the name is called start name here). In the TCD one level up, an outgoing arrow is sought from the corresponding task, which

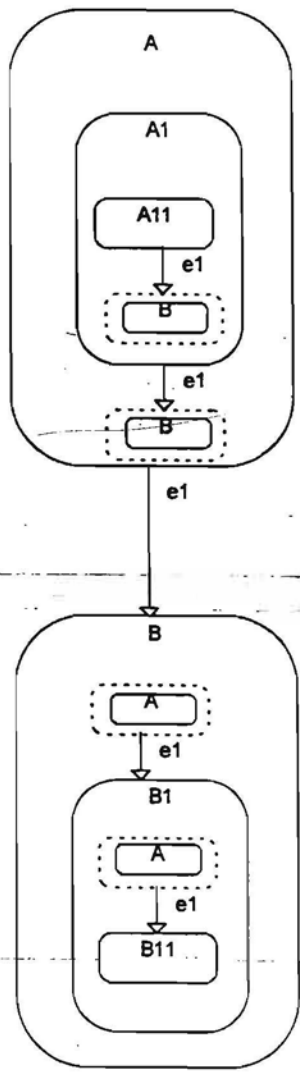


Fig. 6.5 Event routing (shown via several nested TCDs)

- has the same event name (including empty)
- the other end of which is
  - an internal (external) task with its name equal to the start name or
  - a referenced task whose name list contains the start name

The decisions for non-elementary tasks are ignored in "upgoing", only event names and referenced task names are taken into account.

Thus upgoing is repeated until the horizontal link is reached, along which the event is routed. In fig. 6.5, starting from A11 and event e1 and using B as start name, first the task A1 with equally named outgoing referenced task is found, then the horizontal link to B.

The downgoing part of routing is started, using the source task of the horizontal link as the start name (A in fig. 6.5). All referenced tasks in the refinement are found,

- which are associated with the same event and
- whose name list contains the start name,

and a copy of the event is sent via each. If the other end is not elementary, the search is repeated a level down, with the same start name. If other end is elementary, the destination queue for the route is found. In fig. 6.5, using A as the start name, two level downgoing is performed, until B11 is reached.

The described rules are such that using the default naming of referenced tasks proposed by the editors (see 6.4) just the natural link is established (including the case in fig. 6.5).

At any level (except intermediate upgoing ones) the event may be multiplied, with a copy sent along each branch. If no continuation for the event is found, the event is discarded. During syntactic analysis, situations with the abnormal refinement are found and warning messages are issued. Actually, the analysis finds only part of these refinement errors (those described in 6.8), others are found during the preparation for simulation (see 11.2).

6.6 GRAPES-BM model development strategies and tool support for them

The main model development strategy in GRAPES-BM is assumed to be top-down, with the TCD being the main diagram built manually. TSD diagrams are generated automatically most of the time. More precisely, according to this strategy, at first the primary tasks, i.e. their TSDs, are entered. Then for each of these tasks its first level refinement TCD is built. The constituent tasks in this TCD are entered one by one and so are the linking events.

GRADE editors during this process automatically build the corresponding refining TSD diagrams and place them in the model tree, just under the parent task.

When a new TSD is automatically inserted, all relevant information (input events, output events, referenced task symbols with neighbor task names, referenced timers, decisions, connected data stores) is automatically transferred to this TSD. The textual sections of task symbol (triggering, performers, duration) are also copied to the corresponding sections of the task body in TSD. Decision contents is also transferred automatically. Unnamed events are transferred in the same way as their named counterparts

The transfer of this information actually only occurs upon saving the TCD diagram. The automatic transfer may be switched off using editor options, in that case the entire transfer must be performed using a manual transfer command (menu item *Edit/This TCD->TSDs*).

If several occurrences of the same task are present in a TCD, the information in the TSD is summed from all occurrences. In particular, for each event and neighbor, where there is no corresponding referenced task already present, new referenced tasks with appropriate task/event names are inserted (or the referenced task name list is extended). Externality information is also retained in the referenced task (and incoming timers likewise).

In case of several task occurrences care must be taken, that textual sections (those appearing also in the TSD) in all occurrences are identical, since only one (actually the last one) will be copied to the TSD.

When a TCD is being modified (new event arrows inserted, events renamed on existing ones, etc.), the corresponding TSDs are updated automatically. But there may be situations where modifications are not transferred automatically, and then the appropriate manual transfer command (menu item *Edit/TCD-> TSDs*) must be executed. A "global" version of this command (*BM Functions/ TCD->*TSDs) is also available in the model tree window. All these automatic updates include only the augmenting of TSDs with new events, referenced tasks etc. Removing unused events from TSDs (i.e., referenced task - event pairs which have no counterpart in any of the task's occurrences in TCDs) is performed as a manually invoked operation (*BM Functions/Delete events unused in TCDs*) from the model tree window. This operation may be performed either at one TSD level (for the subtree under it) via the subitem *From Subtree TSDs* or for the whole model via subitem *From all TSDs*. In both cases the ET is also cleared of unused events.

The above-mentioned automatic GRADE support ensures automatic correct TSD building when the TCD contains elementary tasks. The only necessary manual operation is extending TSD by task data not available in TCD, like attributes.

Most GRAPES-BM models contain several levels of TCD refinement. The intended strategy is, as soon as one TCD is completed, to refine some of its subordinated tasks by their own TCDs. GRADE editors also provide support here.

The principal idea here is TCD template generation from TSD. When a new refinement TCD for a task is started (the first one or an alternative), the TCD diagram containing all referenced task symbols (along with their names or name lists) and their associated events, data stores and data objects (with access paths) from TSD and one dummy unnamed internal task in the center is generated automatically. Fig. 6.6 shows this template TCD for task T3. Four referenced tasks with the associated events (e1, e2, f1, f2) and the data store are copied (but not the decisions, since they may be quite different in the refinement).

The user then modifies the TCD, inserting more internal/external tasks in it and reconnecting the incoming/outgoing event arrows. Care must be taken not to leave the dummy unnamed task as it was generated, since an unnamed internal task is an error. Referenced task symbols may be replicated if necessary It is also allowed to reshuffle referenced task lists for one fixed incoming or outgoing event into several referenced task symbols in a TCD. New referenced task_names in TCD may be added only in case they are also added in the parent TSD and the corresponding upper TCD is also updated.

Both of the automatic generation features described above in totality support automatic interfaces between TCD levels (according to 6.5) in all normal cases. Namely, no additional referenced task editing is usually necessary, neither in TCDs nor in TSDs, independently of how many TCD refinement levels are used.

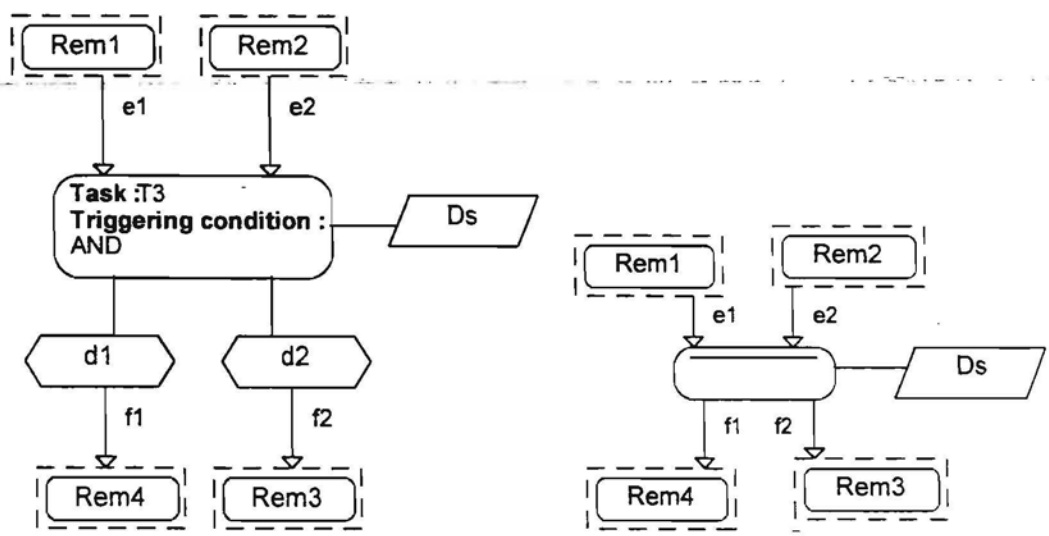


Fig. 6.6. Example of TSD and corresponding TCD template

The next section describes an alternative strategy

6.7 The alternative way: from TSDs to TCD

In addition to the standard refinement method, where every non-elementary TSD is manually refined via its TCD and corresponding subordinate TSDs are generated automatically, another methodology is also available.

There only TSDs are explicitly built and manually placed in the required hierarchy. This hierarchy corresponds to a function decomposition tree in software engineering terminology. When all direct subordinate TSDs for a TSD have been built, a special "Build TCD from TSD" operation may be applied to the parent TSD level. Then the appropriate TCD is generated automatically from all subordinated TSDs, basing on

- incoming/outgoing event names
- names of referenced tasks.
- decisions/outgoing events

Namely, each TSD is converted into an internal task and placed in the generated TCD. Two tasks are linked by an event arrow, if one TSD contains the outgoing part of it and the other the incoming part (as defined by event names/referenced task names).

The non-matching referenced tasks are retained as referenced in TCD (they correspond to links to the next level).

When all incoming/outgoing events and decisions are inserted in the appropriate TSDs of one level, the appropriate TCD can be automatically obtained.

The generation principle does not work when two or more occurrences of a task are supposed to be in a TCD.

A TCD can even be generated when referenced task names are omitted and generation is done based only on matching incoming/outgoing event names. This approach does not guaranty the desired ordering of the TCD.. The names of referenced tasks in TSDs are inserted automatically (according to the generated TCD). Some manual improvement to the generated TCD is sometimes required.

In any case the generated TCD may be further modified manually. Fig. 6.7 presents a set of three TSDs and the generated TCD.

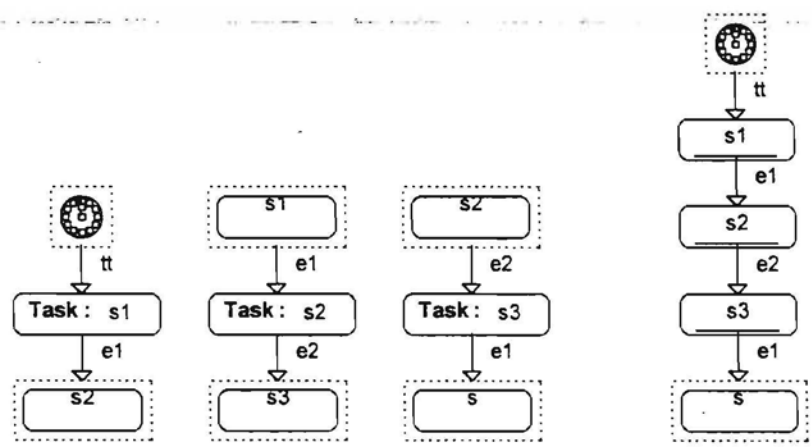


Fig 6.7. TSDs and the generated TCD

6.8 Formal consistency rules between TCD and TSD

The built-in automatic consistency features in the editors normally will guarantee consistent refinements between all TSD and TCD levels, especially if strict top-down design is used. The user is only required to connect all incoming/outgoing referenced tasks presented in TCD templates to some existing tasks in these TCDs. However, complicated diagram updates, especially manual editing of TSDs, may violate this consistency between levels. The syntax analyzer provides two facilities for ensuring this consistency :

- consistency between TSDs and TCDs is checked during the syntactic analysis of TCDs
- consistency between adjacent TCD levels is checked during a special consistency check operation

The following consistency rules between TSDs and TCDs are checked by the syntax analyzerduring the analysis of a TCD:

- rule governing the relationship between a task's TSD and its refinement in a TCD

  all incoming/outgoing events (and control flows) and referenced tasks (and referenced timers) in a TCD must be represented in the TSD. It means that for each pair of incoming events with referenced task names in a TCD, there must an equivalent pair in the TSD. The partitioning of the task name lists into several referenced task symbols is permitted. The same must be true for outgoing events. For decision task TSDs, it is not significant from which of the decisions the relevant pair actually go out. This feature is in line with the assumption that the decision in a non-elementary task is just provided to improve TCD readability. If there are unnamed referenced task symbols in the TCD, then the corresponding unnamed referenced task symbols must also be in the TSD. Violation of any of these rules causes an error during analysis.

- rule between a task's TSD and its occurrences in other TCDs (one or several):

  all incoming/outgoing events/control flows (and incoming timers) and decisions in an occurrence must be present in its TSD. Events must be attached to referenced tasks, whose names (or one of the names in the name list) must coincide with the corresponding neighbor name (i.e., either task name or referenced task name, if the neighbor already is a reference). Certainly, there may be more events/referenced tasks in TSD (corresponding to other occurrences). Violation of the rule leads to an error during analysis.

The other facility - the global consistence checker may be invoked from the model tree window as a separate function, via *BM functions/Consistence checker*. It may be applied either to the *Current TCD* or to *All TCDs*. It checks the following consistency rules between two adjacent TCD levels (where a **child** TCD means a refinement TCD for a task occurrence in the given TCD; and a **parent** TCD means the reverse relation; the task occurrence which defines the parent/child relationship is called the **linking** occurrence) :

- rule between a TCD and its parent TCD:

  an incoming/outgoing link in a TCD (i.e., a pair <incoming/outgoing event, referenced task name>) must have its counterpart in the parent TCD, i.e. an appropriately named incoming/outgoing arrow from the linking task occurrence leading to an appropriately named task or reference. The similar rule must be true for control flows and timer events Violation of the rule leads to a warning since there may be no violation of the rule for another occurrence of the same linking task

- rule between a TCD and its child TCDs:

  each incoming/outgoing event arrow of the linking task occurrence must have a corresponding incoming/outgoing link in the child TCD (i.e., the event path must be continued inside the child TCD).

The similar rule must be true for control flows and timer events. Violation of the rule leads to an error message.

The consistency of parent/child relationship is checked independently for each task occurrence (having a refinement TCD) in the role of the linking occurrence.

The consistency between TCD levels is of great importance also for non-simulatable models since any inconsistent event link is a logical flaw in the model. Therefore consistency checking is recommended also for informal models built for system modeling. The consistency checking can only be applied to a model after syntactic analysis. To facilitate analyzing of non-simulatable models, a special "enlightened" syntactic analysis is available (see 6.9)No consistency requirements are placed on decisions in TSDs and their respective equivalents in refinement TCDs. The same is true for data stores.

Not all event routing irregularities significant for simulation may be found during syntactic analysis or consistency check. Therefore additional routing checks are done and additional warning messages appear during preparation for simulation.

6.9 The syntax for non-simulatable models

The syntactic analysis is of great importance also for non-simulatable models since it helps to reveal essential logical errors in models. The consistency checking between TCD levels which often reveals inconsistent modifications in models is especially valuable. The **consistency checking** can only be performed **after the conventional syntactic analysis** of the model. To facilitate the analysis of informal models, the non-simulatable syntax option is available. To switch this option on, uncheck the *Simulatable syntax* checkbox in the *Options/Settings* dialog box.

The non-simulatable syntax option allows one to place an arbitrary text (without having any error message during analysis) in the following syntactical elements:

- transfer time specifications (in a TCD)
- SET and REPEAT options of output events (both in a TCD and TSD)

These two features together permit one to "decorate" event arrows in a TCD with rich informal comments while keeping the model syntactically correct.

Remark. Do not use non-simulatable syntax option for simulatable models, the simulator will ignore any SET and REPEAT options if you try to do so.

7 Transaction semantics of TCDs

7.1 The concept of the transaction

Very frequently each TCD level of a business model corresponds to a real business function or subfunction performed by the enterprise to be modeled. This is especially true when the structuring is not very deep.

Therefore it seems natural that each complex task corresponds to a transaction - a certain sequence of actions with precisely defined start and end moments. The start/end moments of a transaction are implied by starts/ends of elementary tasks contained in the transaction.

Thus, in GRAPES-BM V3.0 we assume, that each complex TSD by default defines a transaction having the same name as the task itself. Even when a TSD has several refinement alternatives, all these alternatives just determine different behaviors of the same transaction.

Transactions are important for modeling purposes: for better understanding of model behavior and for defining a reasonable semantics of merging several subactivities of the same activity.

On the other hand, they are very significant for simulation, since some of the default statistics for a transaction are the same as for elementary tasks, and they are the basis for efficient use of user defined attributes.

The main syntactic and semantic problem in using transactions is to define how and when elementary tasks start and end the transaction. There are both default and explicitly controlled transaction management facilities. Transactions, like elementary tasks, have instances during execution. Each instance is characterized by its name and a unique system defined **Transaction Identifier (TID)**. TIDs have similar meaning for BM semantics definition like process identifiers (PIDs) have in the SDL language.

7.2 Default behavior of transactions

When a transaction (task) has one level of refinement (i.e., all tasks in the refining TCD are elementary), all tasks (both internal and external) in this TCD constitute the static area for this transaction.

The default start of this transaction is the start of any task (internal or external) in this area, which is triggered only by the following classes of events:

- timers
- events coming from referenced tasks
- spontaneously generated events

If the task triggering condition contains no OR, it can be determined statically whether the task starts a transaction. Otherwise it can be determined only dynamically, for each instance separately.

As soon as a transaction instance is started, a new unique TID value is generated for it. From now on, all event instances circulating within the transaction are tagged by this TID. So are also all task instances belonging to the transaction instance. More precisely, the tag consists of

- transaction (task) name

- TID value

There is no explicit use of TID values in GRAPES-BM, these values are used only in implicit comparisons. Actually these values are integers.

The start task tags all its outgoing event instances (including control flows) with the same tag value.

Tags are not placed on events which leave the area of the transaction (i.e., are directed to outgoing referenced task symbols).

If a (non-start) task within the area is triggered by a simple tagged event, the same tag value is reproduced on all its appropriate outgoing events. If a tagged event is mixed in a triggering condition with non-tagged events (timers, events coming from remote tasks etc.), the output tag value is again this one. The most complicated case is when the triggering condition ANDs several tagged events. Then an implicit **merging condition** is added to (or forms) the WHERE condition:

- if the transaction names are the same, TIDs for all events must coincide
- if transaction names are different, no additional condition is required.

Only if the merging condition is true, the elementary task is actually triggered. The outgoing tag is defined in the natural way (the common value). Merging condition refers also to implicitly ANDed control flow instances (if they have tags).

The merging condition operates on the principle, that only concurrently executed subactivities of the same activity instance should merge together

The unique tag value obtained from the triggering events determines the transaction instance to whom the task instance belongs and also the tags of all the outgoing events.

If there are several levels of TCDs, then each level defines a transaction. When tagged events from a higher level enter the next lower level (via referenced task symbols), the higher level tags are retained by them. If e.g., an entry from a referenced task starts a new transaction of the current level, then events of this transaction carry tag list, corresponding to two adjacent task levels. When task nesting is deep, the tag list can be arbitrarily long. The merge condition requires TID equality for all appropriate levels. Namely, in order for this condition to be true for a set of events, for all events of this set having tags in their lists with one common transaction name, the TIDs in these tags must also be equal.

For a higher level transaction its static area consists of tasks in its TCD, as well as in all subordinated TCDs (of all levels). In the entire transaction area, tags of this level are propagated according to the above-mentioned rules.

Higher level transactions are also started automatically in nearly all desirable cases. The default starting rule is the following. When a lowest level transaction is started (along with an elementary task in it), each event in the triggering set of this task is independently examined:

- for a timer event it is checked whether it comes from a timer symbol in the given TCD. Nothing more is triggered in this case. If, on the contrary, the timer event comes from a referenced timer symbol, the next level transaction is started also. This action is repeated until the TCD level with the actual timer symbol is reached. That level is the highest activated.
- for an event coming from a referenced task the source of this event in the next higher level TCD is checked. If it is a task (internal or external), nothing more is triggered. If, on the contrary, the source is a referenced task, the next level transaction is also started. The action is repeated up to the level, but not including it, where the source of the event is another task.
- for spontaneous events no addition triggering may occur, since they are active only at the elementary level.

Starting the transaction of the corresponding level means generating the appropriate TID. For a while this TID is "resident" only in the lowest level task, but it can return to its home level via events returning to this level. Only one instance of each level transaction is started as a response to triggering the lowest level task (even when there are several events in the triggering set which descend along the same path).

However, if one event in an upper level is multiplied to several copies in the lowest level and each of these copies triggers an instance in the lowest level transaction, then as many instances of the upper level will also be started. Normally such situations should be avoided since these independent transaction instances in the lowest level can never merge (which normally should occur for subactivities of one activity). The best way to avoid starting unnecessary transactions in the lowest (and subsequent) levels is to use NOSTART option (see later) at all lowest level entries, except one.

NOSTART for transaction control prevents the starting of current level transaction and all simultaneous upper ones.

There is also a default dynamic **transaction end condition**. Namely, when there are no more event instances with the given TID value in any of the queues within the area, the transaction with the given name and TID is ended (more precisely, it is ended, when the elementary task consuming the last such event stops or the event is discarded). Any level of a transaction may be ended by default this way.

A special case is triggering conditions containing AND ALL and AND <n> options. Tagged events which are consumed in groups by these options lose their identity and tags after such task (except the case when equally named outputs leave the task, then each tag instance is retained in the corresponding output instance). In addition, events in the group (fixed-size or ALL-group) never participate in the merge condition, i.e. tag comparison is ignored for them. Merge condition refers only to other events in the AND-list.

All "new" output events from a group-triggered task have no tags at all (except the case, when a tagged event participated in the "individual part" of AND ALL, then such a tag is merged and propagated, as for the normal AND).

Spontaneously generated events also have no tags (but they are used as new transaction starters, see above). If untagged "normal" events (i.e., except timers, spontaneous events and events from referenced tasks) trigger a task, no default transaction start occurs. If starting is desirable, an explicit START option must be used.

It should be noted that default transactions of any level always are structurally nested. Namely, if a lower level transaction starts within a higher level one, then it always ends before (or simultaneously with) the higher level transaction. This is ensured by the nature of the default rules.

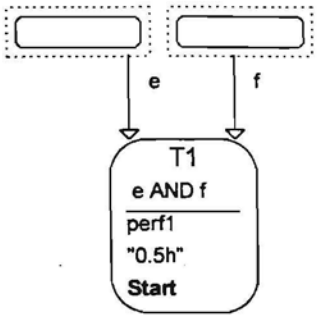
7.3 Transaction control facilities

In addition to default behavior, there are four explicit transaction control facilities:

- START option
- END option
- NOSTART option
- NOTID option

Transaction control options are present only in TCD diagrams (not in TSDs).

START option is placed in task body in TCD diagram (but not in TSD)



The START option may be placed in tasks which otherwise could be default start points of a transaction or in any other task as well. The START option syntax is the following:

START [task_name] {,task_name}*

where task_name is an appropriate task from the model tree (which is an ancestor of the current task).

Omitted task_name means the task in whose refinement we are.

The meaning of the START option is to start a transaction (or transactions) with the specified name(s).

To be more precise, all transaction levels from the lowest one to the highest one specified in the task name list are started. The levels are uniquely found from the model tree. Formally, it would suffice always to mention only the necessary highest level task name, but it is recommend to include all intermediate levels for the sake of readability. This convention is introduced in order to preserve the strict hierarchy of transactions defined by the default rules.

Example of the START option:

START order_entry, order_processing, client_transaction

The explicit START is necessary for three purposes:

- there are several default start points for a transaction from which the proper ones are to be singled out
- transactions of several levels are to be started simultaneously and default rules are insufficient
- transaction covers only part of the TCD.

If there are several possible start points of transaction in a TCD, then two situations may be true:

- all starts are real alternatives how the current transaction could start. In this case no START option is necessary (or all of them could be marked by simply START, just to reveal this fact to the reader). In this case each of the start points starts an independent transaction, which should never merge
- One of the start points starts the transaction (e.g., order entry), while others represent some auxiliary actions (e.g., updating the price list). Then the proper start point should be marked by the START. The auxiliary ones then must contain NOSTART (see later). Then auxiliary actions represent no transactions, events participating there are untagged.

With several levels of refinement in business model, there may be a need to start several transaction levels simultaneously. If default rules are insufficient, the START option with task name list is used.

In general, the START option may be placed in any task , which receives no tagged event of the current (or of that specified in the START option) level. Then the transaction is started with this task, but not with the default start point. Certainly, the default start points of TCDs leading to this marked task then should be

marked by the NOSTART option (see below). This feature allows one to define transactions covering only part of a TCD (an explicit END option then is used to end such transaction, as a rule).

If a task marked by the START option is triggered by an event which already contains a tag of a level specified in the START option, it is reported as an execution error during simulation (in order to avoid a recursion of sorts).

A special option

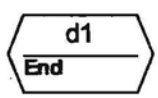
NOSTART

is also available in task symbols. This means that the default start point must not start a new transaction (of the current level and possible higher levels). A typical use of this option could be when a TCD level is defined just because of diagram size, without any functional meaning. Then all potential start points of such a diagram should be marked by NOSTART, in order not to affect the transaction behavior defined by a higher level TCD. Existing higher level TIDs pass through task marked by NOSTART without problems.

The implicit transaction end works well when there are no "junk-events" remaining in queues forever. However, in some cases a normal way of describing a model just requires to leave some events in queues unconsumed. A typical situation is when time-out activities are to be described: then either the unused reminder event remains in a queue, or the too-late message remains unused. To cope with such situations, the explicit **END** option is used.

The option is placed either at the bottom of the body of a task in a TCD

or at the bottom of a decision



(in order to have effect only if this branch is taken).

The textual syntax of END is much the same as that of START;

END [task_name] {,task_name}

The syntax details and defaults are the same as for START, including the set of affected transaction levels.

This option forcibly ends a transaction (or transactions), by emptying all queues in the area from events having tags with the specified name and the current TID value (i.e., the TID value as it would be passed further). The area for emptying is determined from transaction name. The current task sends its events untagged.

In addition, active task instances holding the specified tags are terminated forcibly, without taking any decisions or sending any output events.

Thus the END-option empties all queues in the area from the specified events and the transaction is ended according to the previous definition. If several levels are to be ended simultaneously, the lowest one is ended the first.

If there is no tag available with the name required by the END-option, it is a semantic error.

A typical position for END is an exit-task (a task passing events only to referenced tasks) of a TCD, but it can be placed in any task.

Yet another special option is

NOTID

This option may be placed on event arrows in TCD diagram, beside the event name (or after transfer time, if it is present). It may be placed only on horizontal arrows (i.e., arrows not coming/going to referenced tasks). NOTID doesn't appear in a TSD.

The semantics of NOTID is that no TID of any level is passed along the arrow ,i.e. the event sent along the arrow has no more TIDs at all.

The main use of this option is to prevent merge problems in tasks emulating global variables (i.e. variables common to all transaction instances). Namely the events representing global data and looping back to the same task should be marked by NOTID (see the event *Account* in Fig 7.1, otherwise the global counter task *Summing* wouldn't trigger upon arrival of the event *payment* from the next transaction instance).

NOTID option may cause an implicit end of a transaction if the TID is being canceled in the last event instance of this transaction.

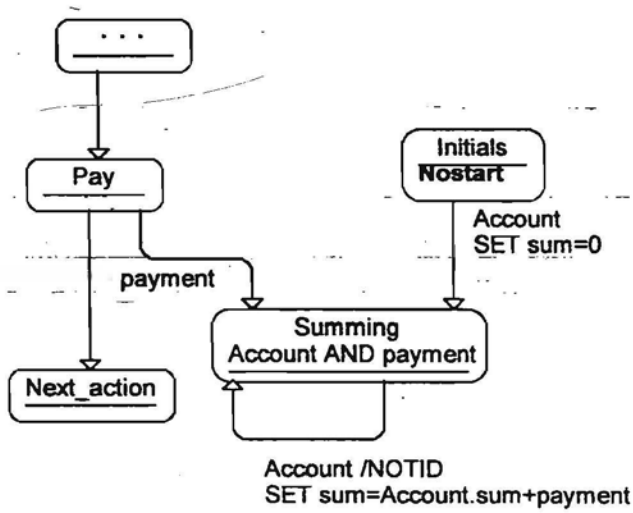


Fig. 7.1. Example of NOTID use

7.4 Attributes of transactions

Since transactions always correspond to normal complex tasks having TSDs, they also have attributes - namely, those defined by the corresponding Type and Attributes sections in the TSD.

Any transaction has the same predefined attributes as any other task - **duration** and **cost.**. Duration specification in TSD for transaction tasks is ignored.

The other numeric attributes of a transaction must be defined explicitly, by supplying their formulas in the Attributes section of the corresponding TSD (or in the ATR table corresponding to the task type). It is

typical, that attributes are only derived here, and they use vertical operations (SUM, AVG, MAX, MIN) on predefined or user-defined attributes of elementary tasks (or lower level transactions). The span of such an operation is the lifetime of the TRANSACTION task instance. Any numeric or duration attributes appearing in any ATR table may be referenced in vertical operations (but not directly!). If an attribute appears in several ATR tables all definitions must have the same type. Partial sums are updated each time an elementary task instance having the specified attribute is ended (within the static area of the given transaction and having the appropriate TID). Thus, if the transaction has attribute *tot_a1* defined by the formula *SUM(a1)*, then for all elementary task (and nested transaction) instances within the given transaction instance the value of *a1* is taken (where it is defined) and summed up. When the transaction task is to be completed, the final values of its (totaling) attributes are passed for processing at a higher transaction level (if there is such). At that moment also the default statistics for the transaction is updated.

Rules of using other attributes from their own ATRs in arithmetic formulas for transactions are the same as for other tasks.

If only attributes of a specific task should be "averaged" at a higher level, it is reasonable to have unique names for them (there is no way to distinguish, e.g., transaction task cost from elementary task cost, no attribute qualification is supported in V.3.0).

The sole use of transaction attributes is for obtaining statistics on them (and/or computing attributes of higher level transactions). There is no way to use transaction attributes to influence system behavior.

8 Additional structuring features of business models

8.1 Interaction of primary tasks

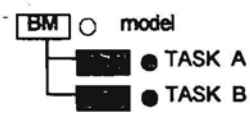
A business model consists of business functions which are represented by top-level tasks (i.e., top-level TSDs). Any top-level task which is refined by a TCD defines its activity which proceeds concurrently with other such ones. Certainly, only these tasks which contain a timer (or spontaneous event) at any of the refinement levels actually become active. By default, top-level tasks function independently of each other.

Sometimes some interaction is necessary also between these top level tasks. GRAPES-BM version 3.0 proposes some facility for describing this interaction. Namely, the TSD of a top level task may contain also internal referenced task symbols containing names of other top level tasks.

Certainly, there may also be other referenced task symbols in top level TSD, but these symbols are necessary for the case when this top level task is "called" somewhere lower, i.e., it has an occurrence there (see next section).

The semantics of the facility are explained on an example.

Let us assume that there are two top level tasks A and B in the model



Let us assume that the task A sends a message e to task B. Then the TSD diagram for the task A must contain elements shown in Fig. 8.1

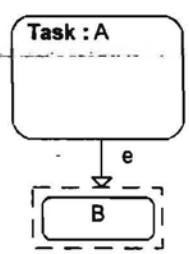


Fig 8.1

TSD for B, in turn, must contain elements shown in Fig. 8.2

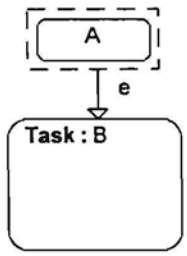


Fig 8.2

In further refinements of both A and B by TCDs top level task names are propagated deeper, i.e., used in referenced task symbols at lower levels, until the real communicating partners in both tasks are reached.

It is forbidden in one top level TSD to communicate directly to components of another top level task, i.e. only top level communication can be defined this way. Lower referenced tasks may appear only for "call situations" (see 8.2).

Such a mechanism is required only to describe top-level interactions. Ordinary TCDs serve to describe interactions in one top level task.

The semantics is such, as if there were one more TCD diagram, containing top-level tasks as elements. Then TSDs for top level tasks would look just this way. However, normally it is unnatural to include such a TCD in a model, since top level TSD represent independent functions.

From the simulation point of view, it should be remembered that in contrast to "calling" top level tasks at a lower level (see section 8.2) no internal copy (in the sense of 11.1) of it is generated, when top level task communication is described via their TSDs. Each top level task (together with its refinement) defines its single static occurrence which behaves independently (or communicates with others using facilities just described).

8.2 Independent tasks and the multiple use of tasks

The standard way of defining a task is to define it as a part of direct refinement of another task. Such tasks are called refinement tasks in what follows. However, it is also possible to push the task definition point (i.e., its TSD diagram) up in the business model tree, thus making it usable repeatedly in several independent TCD diagrams. Such tasks are called independent tasks. Top level tasks are always independent tasks by definition.

There is no difference in TSD syntax whether the task is a refinement task or independent task. In any case the task may be elementary or complex, it may be a transformation or decision task. It should be remembered, that an independent task must have all its input and output events and referenced tasks specified, in order to use it properly in a TCD diagram. For this reason top level TSDs also must have input/output events and referenced tasks specified, when they are used in other TCD diagrams (this reason is different from the one described in 8.1).

Thus the distinction between refinement and independent task is made only by its TSD position in the business model tree.

Any task - refinement or independent may be used in any TCD diagram where it is visible.

The visibility area is defined in a manner typical to GRAPES (see also 1.2) :

- top level task is visible in any TCD within the business model;

- task placed just under a certain task T is visible in any of TCDs placed somewhere under the task T, i.e., either in a direct refinement of T or in a TCD lower in the hierarchy.

It is not allowed to redefine a task defined higher once more at a lower level. At the same time, it is possible to have different definitions of a task with the same name in different branches of the business model tree.

Any task - whether refinement or independent is used ("called") in a TCD diagram, using the same internal task symbol. The task name in this symbol must correspond to a visible task (TSD) name. The input/output connections and decisions (if any) of the symbol must match those specified in its TSD. The event matching is done by their names and referenced task names, control flow being the only event without a name. There may be less connected inputs/outputs and decisions to the symbol than in the corresponding TSD. There is no restriction on how many times a task is used in several TCDs. It is forbidden to reference tasks recursively (e.g., TCD of the task A uses task B and vice versa).

In a strongly top-down design mode, TSDs for independent tasks are defined at the proper position in the model tree using the TSD editor. The referenced task symbols may be left unnamed at first, their names will be supplied later when occurrences appear. However, automatic insertion in TSD when referencing this task in a TCD diagram with more input/output events works also in this case.

When a TSD is generated automatically upon mentioning a new (invisible) internal task in TCD, this TSD is placed in the position of a refinement task. It is possible to push manually this TSD up in the tree, to make it usable in several TCDs. From this moment, all automatic updates in this TSD invoked by the usage of the task in several TCDs are summed up irrespective of the position of TSD (according to visibility rules!).

The intended formal semantics for multiply used ("called") tasks is that of macro-expansion: if the referenced task has a refinement, the internal task symbol is virtually substituted by its refinement TCD, with input/output events connected accordingly.

9 Simulation parameters and their usage

In order to make simulation experiments with a BM model more convenient, a special simulation parameter table has been introduced (SP). This table contains **name, type** and **value** columns, thus defining named constants like those in fig. 9.1

Name:	Type:	Value:	Description:
Task_1_duration	DURATION	"5m"	
Fixed_costs	FLOAT	5.01	
Line_count	INTEGER	20	

Fig. 9.1 Example of SP table

Only the following elementary types

- integer
- float
- duration
- time

may be used in the type column. The additional description column contains any informal information. The value column must contain a valid constant of the given type.

The main use of constants from SP are in TSD diagrams.

These constants may be referenced in nearly all the textual elements of any TSD diagram of the model where a constant of the appropriate type is valid, i.e. they may be used in attribute values or expressions, as duration values, in decision formulas, output event data setting, REPEAT values for output event sending, input event REPETITION specifications and WHERE conditions (but not in PRIORITY, MAX INSTANCES and ALTERNATIVES sections).

The most typical use of simulation parameters is just to set task attribute (predefined or user defined) values, since these are the values which need to be easily changeable during simulation experiments.

The other diagram where these constants may be referenced is in the ORG diagram (performer efficiency), ET table (in timer repetition, transfer time specifications and persistence), ATR table (in default and formula) and TCD diagram (event transfer time and copies of text items in TSD).

During a simulation session the parameters may be both viewed and modified (in a special tabular form). The new values immediately have effect in the session continuation, without any model reanalyzing. During the session these constants may be saved in the repository. Named saving is possible, thus several sets of values can coexist in the repository for simulation experiments. However, there is only one set of values visible via the editor (namely, those set by editor).

10 Data in GRAPES-BM

Though GRAPES-BM is a pure modeling language, some data processing is present in it. In general it is a small subset of GRAPES/4GL facilities, though some specific features are also present. This section describes constants and data expressions as they can appear in various GRAPES-BM constructs.

10.1 Constants

For full use the following type of constants are available in GRAPES-BM:

- integer constants
- float constants
- duration constants
- time constants

Integer constants are unsigned strings of digits (not exceeding Maxint for 4 bytes, i.e. 2147483647). Where allowed signed constants are obtained as constant expressions with unary minus prefixed to an integer constant.

Float constants are in the form

int_const.int_const

also unsigned. Signed float constants again are obtained as expressions. The form .01 is not permitted, use 0.01

Duration constants are strings in double-quotes, containing any descending unit sub-sequence from the following units:

days	(d)
hours	(h)
minutes	(m)
seconds	(s)

Characters in parenthesis show the unit qualifier. Units are separated by a colon. The unit amount is a integer or float constant. For seconds the amount is rounded to an integer.

Years and months are not used in GRAPES-BM constants.

Examples:

"1d"

"3m"

"1d:1h:1m:1s"

"2h:10.25m"

"100.02d"

"1m:10s"

Time constants also are strings in double-quotes. They may be in date or date-time format. Date format contains year. month and day. Date-time format, in addition, contains hour, minute, second. The separator in the date part is period, in the time part - a colon, between parts - just one blank space.

Each unit element is a fixed-format integer, without unit qualifiers:

year:	1900 .. 2099
month:	01 .. 12
day:	01 .. 31
hour:	00 .. 23
minute:	00 .. 59
second:	00 .. 59

Examples

"1996.03.28"

"1996.04.30 12:00:00"

Invalid dates such as 02.30 are converted to valid values, 03.01 or 03.02 in this case.

The following constants may only be used in an ATR table and in the Attributes section of TSD diagrams, for setting constant values of attributes (but not in expressions).

String constant - a string in double-quotes.

List constant - a comma separated list of constant values.

10.2 Data Expressions

There are two kinds of expressions in GRAPES-BM V3.0: the special ones such as event expressions in triggering conditions and performer expressions, and general data expressions.

This section describes general data expressions. They are used in the ATR table, the Attributes section of the TSD body and output detailings.

The following types of data expressions are present in GRAPES-BM V.3.0 :

- integer expressions
- float expressions
- duration expressions
- time expressions

A special type of expression with a different use are Boolean expressions, and they will be defined at the end of this section.

Arguments of an expression may be the following :

- direct constant,
- named constant from SP,
- user-defined attribute_name (from the ATR table which corresponds to the given task type),
- predefined attributes of a task (duration, cost),

- input event_name (if the event has an elementary data type), may be used in expressions within textual elements of a task (in a TSD or TCD),
- input event_name.field_name (if the event has a record data type), may be used in expressions within textual elements of a task (in TSD or TCD)(for nested records qualifications of type field1.field11.field111 are used),
- built-in function.

Expressions are built from arguments using operators, parentheses and vertical operations. Vertical operations : SUM, AVG, MAX, MIN have the form : op(var_name), where var_name is an attribute_name (user defined or predefined, may be used in formulas for transaction task attributes) or an event_name or event_name.field_name(used in formulas inside task triggered by event groups). The span of a vertical operation depends on contexts of the two formula (either all elementary task activations within the transaction instance or all events within the actual activating group). Formally vertical operations may be applied also to a single object (then just the argument value is returned).

The operator priorities and use of parenthesis are the standard ones.

Now some details regarding expressions according to their types. In what follows, by variable we understand an attribute name, event name or event field name, respectively.

Integer expressions may contain

- integer-valued constants or variables
- operators +, -, *, DIV, MOD
- vertical operations SUM, MAX, MIN
- integer random functions
- INTEGER function from real expression (returning the nearest integer, e.g., INTEGER(0.7)=1, INTEGER(0.2)=0, INTEGER(1.2)=1)
- duration_expression DIV duration_expression.

Float expressions may contain

- integer or float-valued constants or variables
- operators +, -, *, /
- vertical operations SUM, MAX, MIN, AVG
- integer or float random functions
- duration_expression / duration_expression.

Duration expressions may contain

- duration-valued constants or variables
- operators +, -
- subexpressions duration*integer, duration*float
- vertical operations SUM, MAX, MIN, AVG
- duration random functions
- subexpressions time - time

Duration expressions must have non-negative values

Time expressions may contain

- time-valued constants or variables
- subexpressions time + duration, time - duration
- vertical operations MIN, MAX, AVG
- time built-in functions NOW and START_TIME

Some restrictions on the time value set will be present (e.g., > 01.01.1900). There are no random time functions (these should be modeled using duration). NOW returns the current model time, START_TIME - the starting point of the simulation session. Both these functions are prohibited in the WHERE part of a triggering condition

Random functions are:

- UNIFORM (min, max)
- NORMAL (mean, deviation)
- EXPONENTIAL (mean)

These functions may be used in conjunction with the integer, float and duration arguments, returning the corresponding type..

And lastly, a description of the Boolean expressions (used in decisions and WHERE conditions).

They are built from:

- relational expressions
- Boolean operators AND, OR (and their alternative notations "&", "|")
- parentheses
- special predicate Is_triggered_by (event), may be used in decisions

Relational expressions are built from integer, float, duration, time expressions and comparison operators :

=, <>, >, <, >=, <=

All operators may be applied to all types, both arguments must have equal types (except that integers and floats may be mixed).

Arithmetical operators have higher priority than comparison operators, comparison operators have higher priority than Boolean operators.

All arguments may be used in decisions, though WHERE has restrictions (see section 5).

String expressions are only of the simplest form: just the direct or a named string constant. They may be only used to set the value of a string attribute (in an ATR table or the Attributes section of a TSD). Strings may not be used in comparisons.

11 GRAPES-BM semantics for simulation

Though the preceding sections already defined the semantics for GRAPES BM V3.0 in more or less, here we present the summary of this semantics, in a more practice oriented way, i.e. in the way this semantics is used in simulation. The informal elements of the language are ignored here.

This section has two purposes. On one hand, it can be treated as an abstract GRAPES-BM execution semantics definition in an operational style. On the other hand, the description is close to the real actions performed by GRADE during preparation for simulation and during simulation itself, including some hints on diagnostics.

11.1 Preparation for execution - tree expansion

After a model has been analyzed and simulation has been selected, first the business model is automatically transformed slightly. The model tree is expanded, under each occurrence of a complex task its complete subtree is attached (where it is not already present). In this expanded tree all occurrences of elementary tasks are found. For each such occurrence an empty queue frame is built. If alternatives are used, they are placed in parallel at the same level.

For each of the named events one queue is built for the occurrence irrespective of how many incoming referenced tasks associated to this event are in the TSD. For each incoming control flow (i.e., for each associated referenced task name) in the TSD, a separate queue is built. These queues are only the potential ones. In the routing phase only those queues will be retained in each occurrence, which have at least one potential source of events (see 11.2), the others are removed.

Queues are built for occurrences of both internal and external tasks. Only for external tasks without names (and without TSD, as a consequence) there are no queues, these tasks are marked as "dead" ones (no events reach them, they send no events, they don't appear in statistics).

Since new internal copies of TCD diagrams are built in this way, which in principle should be observable by the user (in traces, statistics, animation, execution-time inspection), unique qualified names are assigned to them.

When no task occurs more than once in a TCD, and no alternative are used, the simplest qualified name is:

TSD_name1.TSD_name2.TSD_name3.TCD_name

When alternatives are used, alternative names in parenthesis are appended to corresponding TSD names:

TSD_name1(TCD_alternative_1).TSD_name2.TCD_name

When there are several occurrences of a task within a TCD, these occurrences are distinguished by appending the occurrence tags (insertable via TCD editor) or artificial numbers if there are no tags specified for task names:

task1.tag1, task1.tag2

Specific tasks in a TCD are named the same way, inserting the task name (normal or extended) instead of the last TCD name e.g.

TSD_name1.TSD_name2.TSD_name3 .

11.2 Event routing

The next step in the preparation for model execution is finding all possible routes for each event emerging in the model.

For each occurrence of an elementary task in the expanded model tree and for each of its outgoing events (control flow) all possible event routes to other elementary tasks in the expanded tree are found.

This is done according to routing rules in section 6.5. Direct "channels" (which correspond to the found routes) are established for each outgoing event (more precisely, for each pair: event and associated referenced task name), thus preparing a copy of the event (or control flow) to be sent along each of the routes. As explained in 6.5, each route definitely contains just one horizontal link (event arrow connecting two non-remote tasks in a TCD) and possibly the upgoing and/or downgoing link defined via referenced tasks. There is one case without an explicit horizontal link, namely, the connection of two top level tasks defined by appropriate incoming/outgoing referenced tasks in two top level TSDs.

Each route ends in a specific input queue of a task occurrence (but there can also exist routes which terminate in the middle, see later). For named events this queue is determined by event name. For control flows the queue is selected on the basis of the coincidence between the start name (see 6.5) and a name in a referenced task name list. The queue is marked as active as soon as at least one route reaches it.

After the routing process is completed, all potential queues in all task occurrences, which have remained inactive, are discarded. Thus they don't participate in simple AND (and default AND for control flows) triggering conditions.

During the routing process some global routing diagnostics are performed. If there is more than one occurrence of a task, it is completely normal, that in a specific occurrence of this task (or its components) some outgoing routes (determined by pair: event name, referenced task name) are disrupted during routing (they are used in other occurrences, in turn). A symmetric situation is for incoming routes. But if an event remains unconnected altogether, this is considered to be an anomaly. Messages created by routing diagnostics are formulated as warnings. A warning is generated, if

- for the given outgoing event name no route from the task occurrence reaches a destination queue
- no outgoing control flow (if there is such in the TSD) from the occurrence reaches a destination queue
- a potential queue for named event is discarded as unconnected
- the last queue for incoming control flow is discarded

Simulator warnings are displayed in the same way as analyzer warnings. After preparation for simulation these warnings will be attached to the most appropriate symbols in the appropriate TCD diagrams and the diagram's status circle changed to yellow in the model tree (if one or more warnings and no errors are present).. The display of warnings may be switched off (in the model options).

Such warnings would never occur if default referenced task names proposed by editor are retained in TSDs and are really used in refinements of these TSDs by TCDs. However, hand-edited TSDs and incomplete refinements may cause these warnings to appear. Since such a construct may be semantically valid in a model, no errors, but only warnings are generated. It should be noted that this kind of diagnostics is by nature incomplete during local analysis of separate TCDs or TSDs (according to 6.12) and therefore has a global character. The local analysis can reveal routing deficiencies only when the TSDs in the model have no superfluous incoming/outgoing events, i.e. they have not been modified manually too much. The routing checks during preparation are even more powerful than the global consistency checks.

During execution, output events having no valid route are simply discarded without any message.

Now some more notes concerning routing in special cases: timers, unnamed externals, TCD alternatives and complex events.

Timers generate only inputs for tasks. They are either directly linked (by a horizontal link) to one elementary task, or have also the downgoing part of the route (defined by referenced timers) and finally reach one or more elementary tasks. Routing is done based only on timer names. The difference between several equally named timers attached to an elementary task each, and one timer "cascaded" down to several tasks, appears only in one subtle cases with random timers. Each timer symbol occurrence in the expanded tree acts as an independent timer, sending its events according to routes.

Unnamed external tasks (without TSDs!) participate in the routing process (via unnamed external referenced tasks). But they are specially marked as "dead" tasks and events being sent to them are not sent at all.

All TCD alternatives are taken into account during the building of the expanded tree. Alternative expansions are included in the same tree. During routing, alternative routes are also found (more precisely, packages of them). During execution, one route from the package is selected, based an alternative probabilities (default is equal chance, if probabilities are absent).

Complex events may neither be generated nor received by elementary tasks. They are only used to reduce the number of routes in high level TCDs and TSDs. During the routing process (its upgoing part), at some TCD level, an elementary event may be "hidden" into a complex event (using syntax defined in 6.2.12). The routing is continued with the complex event keeping the elementary event name in mind. In the downgoing part of the route the level is found where the original event is singled out again from the complex one and finally reaches its destination. When no event refinement is found the original route is discarded.

11.3 Starting the execution, timers

Now the execution of a model may start. The simulation time is set to the selected start value. In general, the simulation time moves forward, when there are no more simulation steps to be performed in the current time moment.

At the beginning, the only active elements in the model may be timers (and spontaneous events described by similar syntax). Each occurrence of a timer symbol in a TCD is treated as an independent timer. Timers with a time point specification become active according to their description. Interval (i.e., REPETITION) timers become active for the first time, when the specified interval has elapsed from system start. Each timer activity generates a new event instance, which travels and is enqueued according to general rules. By default, timers are instantaneously enabling ("0s" persistence), i.e., if they cannot be used for triggering in the same simulation time moment, they are discarded (when the model time is advanced, thus all simultaneous events actually "meet" the timer). Timer events always are untagged.

If several alternatives of a task (at any level including the top one) contain timers, they all function independently, so no probabilities affect this behavior.

11.4 Starting a task

As soon as a new event is enqueued in queue (for an elementary task), it is ascertained whether the task can be triggered. If it cannot be, nothing is done in the task. If the task can be triggered according to the triggering condition, the task is marked as potentially triggerable and looks for performers. The merge condition is always an integral part of the triggering condition if incoming events have TIDs. If one of the appropriate performer sets is found to be free and available, the task is triggered. Otherwise the task is put on the waiting list for performers (the relevant ones). As soon as some of the relevant performers are released (or become available) a new test for triggering is done. Priorities are taken into account in this test, but within one priority level all waiting tasks have the same chance to be triggered (the precise scheduling is implementation dependent). If any of the involved events has a limited persistence (e.g., a timer), it is

removed from the queue at an appropriate model time moment. This can make the task untriggerable and thus removed from the waiting list.

The actual triggering set of events is consumed (transferred from queues to the task instance data). Timers are processed according to rules in section 4. The selected performer set is assigned to the instance. Performer availability may expire before the task is completed, but this fact is ignored in version 3.0.

Just before starting, the following actions are performed:

- the current value of duration is computed (possibly using consumed events as inputs and taking into account the minimum efficiency of selected performers)
- if the task is a transaction start (default or with explicit START option), a transaction(s) is started - a new TID for the required level is generated and the tag element created (or appended to inherited tags)
- Only then the task is started. From the technical simulation point of view, an active task does nothing. It only waits for its duration to expire.

Each task instance has its unique Id, which is used only to identify instance-related actions in the trace. It may also be part of one or several transactions at different levels, and then the task instance carries the corresponding TID values.

11.5 Ending a task

When the time point where a task ends is reached, the following is done

- its cost is computed, according to the formula

  *duration*SUM(Cost_per_hour)*

  for all performers used (duration converted to hour units), taking into account efficiency as well if used
- user defined attributes are computed according to their formulas. If a random value is used somewhere, each occurrence is a separate random generator
- if the task is a transformation task, all outgoing event (according to TSD) instances are created with their data set by SET options and sent (see next section). Each created event instance has its unique ID (used in trace). The data are passed or set according to section 5.8. Tags are added according to section 7. Group sending (for multiple triggering) and REPEAT option generates a group of independent events.
- if the task is a decision task, the decisions are evaluated one by one, and these found valid are executed (the corresponding events sent)
- if the task is in a transaction, whose attribute formulas reference attributes of the given task, the appropriate partial sums are updated
- if the tasks ends a transaction (by default or forcibly), the transaction instance attributes are evaluated and passed up (to a higher transaction)
- the task statistics for the task occurrence(and terminated transactions) are updated.

All task ending activities are performed as a group, without advancing time. Only then the task is really ended.

11.6 Sending an event

A generated event (with or without data) is sent according to the following rules:

- all valid (i.e., connected to a queue) routes for the event in the given occurrence are found and a copy of the event is sent along each. If there are TCD alternatives involved, one route from the package (see 11.2) is selected randomly according to the probabilities. If the route leads to a "dead" task, the event is sent nowhere and ignored in statistics. If statistics are required, a named external task must be used.
- the transfer time for each route is found. Either the transfer time of the single "horizontal" link in the route is taken (if it is specified in the TCD) or the transfer time from the ET is taken. If nothing is specified, the default is zero. Links to incoming/outgoing referenced tasks never affect transfer time (it is forbidden to specify transfer time for them in a TCD).
- When the transfer time has elapsed the event is enqueued in the destination queue.

The whole sequence of simulation steps is repeated until the end of the simulation session..

12 Simulation statistics

12.1 General principles of automatic statistics gathering

The GRADE simulator, during execution of GRAPES-BM models, supports automatic gathering of statistics about the model execution. There is a list of predefined statistics, which can be gathered by switching them on in the simulator control window. Additional statistical items may be included on the basis of user defined task attributes.

Though the gathering of any statistics item may be switched on or off, the formulas and gathering rules for each of the items are predefined and may not be modified by the user.

The following groups of statistics items are available:

- statistics on tasks
- statistics on performers
- statistics on events

Statistics on tasks are gathered on every occurrence of an elementary task, and, in addition, on every occurrence of a complex task which defines a transaction. Complex tasks which define no transactions are ignored in statistics. For elementary tasks all possible items are available, for transactions only some of them.

Both for elementary tasks and transactions, any user defined attribute having numeric or duration type may be used to define additional statistics items. The processing is similar to that for the predefined attribute *cost*.

The gathered statistics are visible in the simulator, and in a special GRADE component named the *Trace Browser*, in the form of tables and EXCEL-like charts. Each table contains a group of closely related statistics items as its columns. It is typical that there are Total, MAX, MIN and Average columns for the same table. Each row of the table corresponds to a task occurrence in the expanded model tree (see 11.1). Each task occurrence is identified by its

- task name

- TCD name

- full qualified task name in the form TSD_name1.TSD_name2.TSD_name3. In the case of several occurrences tags (user defined or generated) are also used (see more in 11.1). Qualified names are necessary if task names are not unique.

Statistics on performers are based on the performer elementsin the ORG diagram of the model. Actually statistics are only gathered for those ORG elements which are referenced in at least one performer expression of an elementary task.

Statistics on events are based on input event queues of elementary tasks. There is a table entry for each elementary task occurrence in the expanded tree, and each event which has an input queue for this occurrence.

Activation, time without any instance, processing time, costs and user defined attributes are defined both for elementary tasks and transactions. The other tables are defined only for elementary tasks.

Table caption	Columns	Variable	Mode
Activation of the tasks	Task name		
	TCD Name		
	Total count of activation	**TOTCA**	2
	Total count of completion	**TOTCC**	3
	Maximum count of concurrently active instances	**MAXCA**	1
	Average count of concurrently active instances	**AVGCA**	1
	Minimum count of concurrently active instances	**MINCA**	1
	Tasks remaining active	**TRA**	1
Task waiting time for start	Task name		
	TCD name		
	Total waiting time for task start	**TOTWTC**	2
	Maximum waiting time for task start	**MAXWTC**	2
	Average waiting time for task start	**AVGWTC**	2
	Minimum waiting time for task start	**MINWTC**	2
Task waiting for triggering condition completion	Task name		
	TCD name		
	Total time for triggering condition completion	**TOTTCC**	2
	Maximum time for triggering condition completion	**MAXTCC**	2
	Average time for triggering condition completion	**AVGTCC**	2
	Minimum time for triggering condition completion	**MINTCC**	2
Task waiting for any performer	Task name		
	TCD name		
	Total waiting for any performer	**TOTWP**	2
	Maximum waiting for any performer	**MAXWP**	2
	Average waiting for any performer	**AVGWP**	2
	Minimum waiting for any performer	**MINWP**	2
Time without any active instance of the task	Task name		
	TCD name		
	Total time without any active instance of the task	**TOTIA**	1
	Maximum time without any active instance of the		

	task	MAXIA	1
	Average time without any active instance of the task		
	Minimum time without any active instance of the task	AVGIA	1
		MINIA	1
Task processing time	Task name		
	TCD name		
	Total processing time	TOTPT	2/3
	Maximum processing time	MAXPT	2/3
	Average processing time	AVGPT	2/3
	Minimum processing time	MINPT	2/3
Task costs	Task name		
	TCD name		
	Total cost	TOTCOST	3
	Maximum cost	MAXCOST	3
	Average cost	AVGCOST	3
	Minimum cost	MINCOST	3
User defined task attributes	Task name		
	TCD name		
	Attribute Name		
	Total	TOTATTR	3
	Maximum	MAXATTR	3
	Average	AVGATTR	3
	Minimum	MINATTR	3

Definitions of variables

TOTCA - Total count of task starts since end of warm-up period. The current value is updated when each instance of this task is started.

TNOW - The current value of simulation time.

TWARMUP - The end of the warm-up period.

NCA - The current number of concurrently active instances of the task. It is set to its actual value at the end of warm-up and maintained after that.

TSLCA - Time since last task start/end (or since TWARMUP for the first start/end).

CCAT - Cumulative concurrent activations_time. It is calculated after tasks start or end as **NCA** multiplied by the value of **TSLCA** and the result is then added to the current value of **CCAT**. At session end, the last **NCA** multiplied by the last time interval is added.

MAXCA - Maximum value of **NCA**.

AVGCA - Average count of concurrent active instances. It is calculated by dividing **CCAT** by the current value of **(TNOW-TWARMUP)**.

MINCA - Minimum value of **NCA**.

TOTCC - Total count of task instance completions in the accounting period.

TRA - Number of instances remaining active (at the moment when statistics are taken).

WTC - Time interval between the occurrence of the previous and current task instance starts (or between **TWARMUP** and start for the first start after warm-up). If several instances of a task start simultaneously, they are accounted in **NTC**, and **WTC** with a zero value added for each.

NTC - Number of task starts, actually the same as **TOTCA**.

TOTWTC - Total waiting time of a task for start. This accumulates from the end of the warm-up period until the end of the accounting period, i.e.

$$\mathbf{TOTWTC} = \sum_{i=1}^{NTC} \mathbf{WTC}_i$$

MAXWTC - Maximum of **WTC**.

$$\mathbf{AVGWTC} = \frac{\sum_{i=1}^{NTC} \mathbf{WTC}_i}{\mathbf{NTC}}$$ This formula is valid with NTC > 0.

MINWTC - Minimum of **WTC**;

TCC - Time interval between the moment when the first event which satisfies the triggering condition arrives and the moment when the triggering condition is fulfilled. More formally, it is the interval between the youngest and oldest event enqueueing in the event set, which actually triggers the task instance and is consumed by it. It is taken into account for gathering statistics, when the instance starts. Remember that it is the interval between enqueueings, and not between enqueueing and start. **TCC** is zero when only one event triggers a task.

NTCC - Number of task starts, actually the same as **TOTCA**.

TOTTCC - Total waiting time of task for triggering condition completion This accumulates from the end of the warm-up period until the end of the accounting period, i.e.

$$\mathbf{TOTTCC} = \sum_{i=1}^{NTCC} \mathbf{TCC}_i$$

MAXTCC - Maximum of **TCC**.

$$AVGTCC = \frac{\sum_{i=1}^{NTCC} TCC_i}{NTCC}$$ This formula is valid with NTCC > 0.

MINTCC - Minimum of TCC.

WP - Time interval between the youngest triggering event enqueued and the moment the task started. It expresses the time waiting for available performers after the triggering condition is true, and MAX INSTANCES may influence the result as well. For a task starting soon after warm-up, it should be noted that the whole interval between enqueueing and start is taken, not only the portion within the accounting period.

NTP - Number of task starts, actually the same as TOTCA.

TOTWP - Total task waiting time for performers availability. This accumulates from the end of the warm-up period until the end of the accounting period, i.e..

$$TOTWP = \sum_{i=1}^{NTP} WP_i$$

MAXWP - Maximum of WP.

$$AVGWP = \frac{\sum_{i=1}^{NTP} WP_i}{NTP}$$ This formula is valid with NTP > 0.

MINWP - Minimum of WP.

The task status is set to "Inactive" when the number of active instances of a task is 0, otherwise - it is "Active" .

TI - Time moment, when task status changes from "Active" to "Inactive".

TA - Time moment, when task status changes from "Inactive" to "Active".

TIA - Current inactivity interval of the task. **TIA = TA - TI**. For the first and last intervals in the accounting period, only the part overlapping the accounting period is taken.

NTIA - Number of intervals of task inactivity (i.e. the number of inactivity intervals in the accounting period).

TOTIA - Total task inactivity time. This accumulates from the end of the warm-up period until the end of the accounting period, i.e..

$$TOTIA = \sum_{i=1}^{NTIA} TIA_i$$

MAXIA - Maximum of TIA.

$$\mathbf{AVGIA} = \frac{\sum_{i=1}^{NTIA} TIA_i}{NTIA}$$ This formula is valid with NTIA > 0.

MINIA - Minimum of **TIA**.

Processing time is computed in a different manner for elementary tasks and transactions.

For elementary tasks

- the counting mode is 2, i.e. all task instances starting in the accounting period are counted, namely, at the moment when they start;

- the processing time is simply the duration attribute value for the instance (if it is defined by a formula, it is always evaluated the start). Even if the task end is after the session end, the complete duration is taken.

For transactions

- the counting mode is 3, i.e. all transaction instances ending in the accounting period are counted, at the moment, when they end

- the processing time is the interval between the instance end and start (even if the instance has started before warm-up), this value is also the duration value for transactions.

It should be noted, that such a definition yields the expected average values of instance time.

PT - Current processing time of the instance (see above).

TOTPT - Total processing time of task.

$$\mathbf{TOTPT} = \sum_{i=1}^{TOTCA} PT_i$$ (more precisely, the upper index in **TOTCA** for elementary tasks and **TOTCC** for transactions)

MAXPT - Maximum of **PT**.

$$\mathbf{AVGPT} = \frac{\sum_{i=1}^{TOTCA} PT_i}{TOTCA}$$ (for elementary tasks). For transactions, **TOTCC** is used instead. This formula is valid with TOTCA > 0 (TOTCC>0, respectively)..

MINPT - Minimum of **PT**.

COST - Cost of current task instance (taken as defined by the language semantics for elementary tasks and transactions, respectively).

TOTCOST - Total costs of a task. This accumulates from the end of the warm-up period for tasks ending in the accounting period.

$$TOTCOST = \sum_{i=1}^{TOTCC} COST_i$$

MAXCOST - Maximum of **COST**.

$$AVGCOST = \frac{\sum_{i=1}^{TOTCC} COST_i}{TOTCC}$$ This formula is valid with TOTCC > 0.

MINCOST - Minimum of **COST**.

ATTR - The value of attribute *attr* of the current task instance (elementary or transaction), the actual attribute name is visible in the corresponding table column. It is any of the user defined task attributes having a numeric or Duration type. Attributes of tasks or transactions to be processed in this way are defined within session parameters. All attributes selected in session parameters for default processing appear in the same table, the attribute name is just one of the columns. A display of the value of one attribute for all tasks may be obtained via appropriate ordering.

The attribute values are computed at the instance end, both for elementary tasks and transactions. For transaction attributes involving vertical operations (see 7.4), their internal accumulation is completed at that moment and the obtained value is passed for statistics processing.

TOTATTR - Total of **ATTR** of task. This accumulates from the end of the warm-up, for all instances ending in the accounting period.

$$TOTATTR = \sum_{i=1}^{TOTCC} ATTR_i$$ Only the defined (i.e., non-NULL) values are accumulated.

MAXATTR - Maximum of **ATTR**

$$AVGATTR = \frac{\sum_{i=1}^{TOTCC} ATTR_i}{TOTCC}$$ This formula is valid with TOTCC > 0.

MINATTR - Minimum of **ATTR**.

12.4 Statistics on performers

This kind of statistic is computed for each separate element of the ORG diagram. In the case of a subtree in ORG referenced more than once, the ORG diagram is considered to be expanded in the standard way. Qualified names are available to distinguish all element occurrences.

Actually only these ORG elements define a row in the statistics table, which are referenced at least once in a performer expression of an elementary task. It is so because only these performers have had a chance to be used for a task. In particular, it means that organizational units, which at best, appear as performers for high level complex tasks, as a rule will not appear in statistics table.

Statistics items are defined in a style similar to that for task statistics, using formal variables.

The previously mentioned modes have a similar meaning for performers as for tasks. Formally for performers the dynamics, idle/usage time and performer seizing/releasing, play the role of task start/end.

But these two kinds of activities are always uniquely coupled, so one can think also in terms of task start/end for performer statistics modes.

When speaking of performers utilization, average and minimum seized instances and idle time, only the availability periods of the given performer, which lie inside the accounting period, are taken into account. The periods where the performer is unavailable are simply excluded from statistics. Performers are counted only for elementary tasks.

Table caption	Columns	Variable	Mode
Dynamic of Performers	Performer		
	Available number of instances	**AVLNP**	1
	Total number of times seized	**TOTSE**	1
	Maximum of simultaneously seized instances	**MAXPI**	1
	Average of simultaneously seized instances	**AVGPI**	1
	Minimum of simultaneously seized instances	**MINPI**	1
	Performers utilization (%)	**UTILP**	1
	Productive performers utilization (%)	**UTILPP**	1
Waiting time of the tasks for the performer	Performer		
	Total waiting time of tasks for the performer	**TOTTW**	2
	Maximum waiting time of tasks for the performer	**MAXTW**	2
	Average waiting time of tasks for the performer	**AVGTW**	2
	Minimum waiting time of tasks for the performer	**MINTW**	2
Performers idle/usage time	Performer		
	Total idle time	**TOTITP**	1
	Maximum idle time	**MAXITP**	1
	Average idle time	**AVGITP**	1
	Minimum idle time	**MINITP**	1
	Total usage time	**TOTUS**	1
	Maximum usage time	**MAXUS**	1
	Average usage time	**AVGUS**	1
	Minimum usage time	**MINUS**	1

Table 1

AVLNP - This value is defined in the GRAPES-BM ORG-Diagram. "Infinite" value is implied for multiple performers without a number specification. For single performer the value is one.

TAV - total availability time for the performer inside the accounting interval.

NCUI - The current number of performers used. It is set to the actual value at TWARMUP.

TSR - The time expired since the last performer seizing or releasing.

CPI - Cumulative performer_time. It is calculated after each performer seizure or release via multiplying **NCUI** by the value of **TSR** and adding the result to the current value of **CPI**. In the same way as for task activations, for the first seize/release after TWARMUP the shortened **TSR** value is used, and at end of session the last special interval is used.

AVGPI - It is calculated by dividing **CPI** by the current value of **TAV**. Namely this way the averaging occurs only over availability periods.

MINPI - It is the minimal value of **NCUI** since the end of warm-up period. Only values of **NCUI** during availability periods are taken into account.

MAXPI - It is the maximal value of **NCUI** since the end of warm-up period.

TW - The current time interval between the moment a task's triggering condition becomes TRUE and when the task's performer expression becomes TRUE. More formally, it is the interval between the actual task start moment and the "youngest" event enqueueing time in the event set triggering the given task instance. The value of **TW** is the same as **WP** in task statistics - waiting for performers. Only the derivation of **TW** is quite different. It is gathered for performers actually seized for the task instance, i.e., in case of OR in the performer expression, the other possible performers don't participate in the statistics. But the accumulation moments are the same as for tasks - each start of the task in the accounting period.

TOTCP - number of seizures of the performer, i.e. the number of times within the accounting period, when the performer participated in a task start.

TOTTW - Total task waiting time for performers to become available This accumulates from the end of the warm-up period until the end of the accounting period, i.e.

$$\mathbf{TOTTW} = \sum_{i=1}^{TOTCP} \mathbf{TW}_i$$

MAXTW - Maximum of **TW**.

$$\mathbf{AVGTW} = \frac{\sum_{i=1}^{TOTCA} \mathbf{TW}_i}{\mathbf{TOTCP}}$$ This formula is valid with TOTCP > 0.

MINTW - Minimum of **TW**.

A performer which corresponds to a multiple element in the ORG diagram, actually represents a group of non-distinguishable performer instances which may be allocated to one or more tasks. When a performer instance is allocated to a task, its status is changed from "Idle" to "Busy". The number of available performer instances is specified in the ORG-Diagram. For a single performer there is only one instance.

T8 - Time moment, when the last "Busy" performer instance status is set to "Idle", i.e. all instances of this performer element become free.

T9 - Time moment, when the first "Idle" performer instance status is set to "Busy".

D10 - the length of the unavailability period for the given performer between **T8** and **T9**. If the performer is available from **T8** to **T9**, then zero.

TPI - Current idle time of performer. **TPI = T9 - T8 - D10** (i.e., the period when none of instances is busy, but the performer is available).

NPI - Number of intervals of performers inactivity.

TOTITP - This accumulates from the end of the warm-up period until the end of the session.

$$TOTITP = \sum_{i=1}^{NPI} TPI_i$$

MAXITP - Maximum of **TPI.**

$$AVGITP = \frac{\sum_{i=1}^{NPI} TPI_i}{NPI}$$ This formula is valid with NPI > 0.

MINITP - Minimum of **TPI.**

T3 - Time moment, when a task instance starts.

T4 - Time moment, when a task instance ends

TT - Processing time for a task to which the current performer is allocated, **TT = T4 - T3** (the same as **PT**)

NTA - Number of intervals of performers usage.

TOTUS - Total performer usage time. This accumulates from the end of the warm-up period until the end of the session. The given **TT** is accumulated for any performer which is actually used for the given task instance.

$$TOTUS = \sum_{i=1}^{NTA} TT_i$$

MAXUS - Maximum of **TT.**

$$AVGUS = \frac{\sum_{i=1}^{NTA} TT_i}{NTA}$$ This formula is valid with NTA > 0.

MINUS - Minimum of **TT.**

$$UTIlP = \frac{TOTUS}{TAV * AV\ NP} * 1$$

This formula is valid for performers whose number of available instances is specified in the ORG diagram(including single performers), otherwise **UTILP** has NULL value.

UTILPP is similar but takes into account also FOR percentages, i.e., each **TT** is multiplied by the corresponding FOR-percentage during the gathering of **UTILPP**.

12.5 Statistics on events

All automatic statistics on events in GRAPES-BM is related to input queues. There are statistics on

- length of queues
- event location time in queues
- intervals between event arrivals.

No special statistics are available on event sending, since for each task sending an event there is a task receiving this event.

There is a table row in the statistical reports for each existing event queue in the expanded model tree (sec. 11.1 and 11.2), i.e. the table row is uniquely determined by

- TCD name
- task name
- qualified task name (like as far task statistics)
- event name

The qualified task name is necessary in case of several occurrences.

The modes for event statistics have different meaning - event arrival in the queue plays the role of task start, and event departure - that of the task end.

Table caption	Columns	Variable	Mode
Lengths of queues of events	Task name		
	TCD name		
	Event name		
	Maximum queue length	MAXQL	1
	Average queue length	AVGQL	1
	Minimum queue length	MINQL	1
Events location time in the queue	Task name		
	TCD name		
	Event name		
	Total events arrived	TEA	2
	Total events left	TEL	3
	Maximum event location time in the queue	MAXELT	1
	Average events location time in the queue	AVGELT	1
	Minimum events location time in the queue	MINELT	1
Time intervals between event arrivals in queue	Task name		
	TCD name		
	Event name		
	Events count	EC	2
	Maximum of time intervals between events	MAXINT	2
	Average of time intervals between events	AVGINT	2
	Minimum of time intervals between events	MININT	2

NEIQ - The current number of events in the queue. It is set to its actual value at TWARMUP.

TSLQE - The time since last queue activity. It is the time expired since the last event arrival or departure in/from queue (or since TWARMUP for the first queue activity).

CETIQ - Cumulative Event-time in queue. It is calculated after event arrival or departure as **NEIQ** multiplied by value of **TSLQE** and the result is added to the current value of **CETIQ.**

At session end the last **NEIQ** multiplied by the last time interval is added.

AVGQL - Average queue length. It is calculated by dividing **CETIQ** by the current value of **(TNOW-TWARMUP).**

MINQL - Minimum queue length. The minimal value of **NEIQ** since the end of warm-up period.

MAXQL - Maximum queue length. The maximal value of **NEIQ** since the end of warm-up period.

TEA - Total number of events arrived since end of warm-up period.

TEL - Total number of events that have left the queue since end of warm-up period.

EAT - Event arrival time in queue.

EDT - Event departure time in queue.

ELT - Event Location time in queue. **ELT = EDT - EAT.**

AVGELT - Average event location time in the queue. It is calculated by dividing **CETIQ** by the current value of **TEA.** This formula is valid with TEA > 0. The computed average value completely corresponds to the expected average event location in queue, when there are few events in the queue at TWARMUP moment and few at session end. The value is reasonable also in cases where there are many events remaining in queue at session end. But the value of **AVGELT** may be higher than the intuitive value when a significant amount of events are in queue at TWARMUP. (There is no ideal formula for all cases).

MINELT - Minimum events location time in the queue. The minimal value of **ELT** since the end of warm-up period.

MAXELT - Maximum event location time in the queue. The maximal value of **ELT** since the end of warm-up period.

LET - Last event time;

CET - Current event time;

INT = CET - LET; (for the first event after TWARMUP **INT = CET - TWARMUP**)

MAXINT - Maximal of **INT**;

$$\mathbf{AVGINT} = \frac{\sum_{i=1}^{EC-1} INT_i}{EC-1};$$ This formula is valid with EC > 1.

MININT - Minimal of **INT**.

EC - total events arrived in the accounting period

12.6 Use of transactions for user defined statistics

Currently these statistics are predefined. Only the statistics items corresponding to the predefined formulas may be obtained. For example, there is no way to obtain an empirical distribution of some task attribute value, or the graph of some variable over time. The only way to define some non-standard processing is via

transaction attributes (see 7.4). Their formulas may reference via vertical operations the selected attributes of elementary tasks in an arbitrary way. Certainly, all other values must be transformed to task attributes beforehand.

But the only way to use the obtained transaction attributes again is to apply default task attribute statistics to them (total, max, min, avg, see 12.3).

13 INDEX

A

C

E

I

N

T

V