31730

p. 8

SCOSII OL: A Dedicated Language for Mission Operations

Andrea Baldi, ESA/ESOC/FCSD
Dennis Elgaard, CRI
Steen Lynenskjold, CRI
Mauro Pecchioli, ESA/ESOC/MOD

Abstract

The Spacecraft Control and Operations System II (SCOSII) is the new generation of Mission Control System (MCS) to be used at ESOC. The system is generic because it offers a collection of standard functions configured through a database upon which a dedicated MCS is established for a given mission.

An integral component of SCOSII is the support of a dedicated Operations Language (OL). The spacecraft operation engineers edit - test - validate and install OL scripts as part of the configuration of the system with e.g. expressions for computing derived parameters and procedures for performing flight operations, all without involvement of software support engineers.

A layered approach has been adopted for the implementation centred around the explicit representation of a data model. The data model is object-oriented defining the structure of the objects in terms of attributes (data) and services (functions) which can be accessed by the OL.

SCOSII supports the creation of a mission model. System elements as e.g. a gyro are explicit, as are the attributes which describe them and the services they provide. The data model driven approach makes it possible to take immediate advantage of this higher-level of abstraction, without requiring expansion of the language.

This article describes the background and context leading to the OL, concepts, language facilities, implementation, status and conclusions found so far.

Introduction

The *need* for the SCOSII OL has matured through the long experiences ESOC have had with the use of configurable generic MCS's. As any other previous ESOC MCS, SCOSII will be configured through databases containing the mission specific knowledge.

This knowledge will not only need to be efficiently defined, but also validated and then maintained, due to the pre-launch test results and/or the frequent changes which do occur during the lifecycle of a mission.

The SCOSII OL concept is designed to augment the traditional ways an operation engineer specifies mission specific configuration data to cover as well knowledge which is *algorithmic* or *procedural* in nature. Thus it is essential to support the operations engineer in:

- specifying and maintaining the mission knowledge in a natural, concise and intelligible manner - without requiring a detailed software understanding or support of software engineers;
- defining the mission knowledge in context-specific dedicated environments, whereby both the HCI and the allocated constructs are specifically designed for each particular information type;
- validating the specified knowledge by means of 'on-line' checks and testing capabilities.

Andrea Baldi (abaldi@esoc.bitnet) works within the Flight Control Systems Department at the European Space Operations Centre (ESOC), Robert Bosch Strasse 5, D-64293 Darmstadt, Germany. Dennis Elgaard (delgaard@esoc.bitnet) and Steen Lynenskjold (steen@acm.org) both work for Computer Resources International, Bregnerødvej 144, DK-3460 Birkerød, Denmark. Mauro Pecchioli (mpecchio@esoc.bitnet) works within the Mission Operations Department at ESOC.

The work described in this article was carried out at ESOC under a contract with the European Space Agency.

Background and Context

For any mission has been the demand to *derive* information from the format which is provided through the spacecraft telemetry parameters. The most frequently used derivation is that of applying a (linear) calibration to convert raw values into engineering units. The calibration is defined by providing value pairs as part of the database configuration.

Although calibrations satisfy a large percentage of the derivation needs, they do not provide a sufficient mechanism as there is as well a need to compute *derived* values by combining other values using an algorithmic transformation.

In the Multi-Spacecraft Support System (MSSS) these algorithms were specified on paper by an operations engineer and subsequently coded by a software engineer. In SCOSI the operations engineer writes the algorithm directly in FORTRAN expanded with a few syntactical constructs to e.g. reference a previous value of a parameter. In both cases the resulting FORTRAN code is compiled and linked with the operational control system software. An error in the algorithm will not be detected before a run-time crash occurs. The turnaround time for changes has from an operational perspective a significant and unwanted delay. Neither systems support version and configuration control functions.

The Spacecraft Performance and Evaluation System (SPES) offers a significant improvement as it allows the users through a dedicated language to define expressions, compute averages, etc. SPES is however limited to work in an off-line context on historical values and has no integration with the control system as such.

The possible largest driver for the requirements is the wish to formalise and incorporate executable operation procedures written in the OL within SCOSII. Whereas algorithms for derived values do not necessary have to be explicit in the run-time context, procedures do have to: one property of a procedure is its interactive nature involving a close dialogue with a human operator through a procedure execution display.

Within ESA, check-out systems have for some time provided capabilities of defining test procedures through special languages; the most significant ones being ETOL (ESA Test Operations Language), ref. [10], and ELISA (Extended Language for Instrument and Spacecraft AIV), ref. [9]. These check-out languages focus on regression testing capabilities.

Two ESOC studies have demonstrated the feasibility of executable procedures within control systems, namely the Expert Operator's Associate (EOA) study, ref. [12], and the Meteosat WorkStation (MWS) study, ref. [13] - the latter now being used operationally. Both projects focused on the internal representation of procedures and the interactive nature of their execution with close coupling to the human spacecraft operator.

The User Terminal Study at ESTEC, ref. [8], has shown the advantages of an object-oriented language in combination with a mission model. The User Language Study at ESOC, ref. [7], was initiated with the purpose of providing inputs to the SCOSII OL and has proven a number of concepts; in particular the advantages of a layered implementation centred around the explicit representation of a data model. Both studies focused on the configurability aspects of the system and associated language capabilities.

From a technological view the existence of powerful UNIX utilities such as lex and yacc, the ideas behind database languages as SQL, advances in workstation performance, and the maturity of object-oriented concepts have further made it possible to implement the OL.

SCOSII, ref. [1][2][3][4][5][6][14], is the new generation of generic control systems to be taken into use at ESOC; the first client missions being Huygens (97), Artemis (97) and

Envisat (98). SCOSII is a distributed control system running on powerful UNIX workstations connected through a local area network. SCOSII has been engineered for high performance throughput; in particular to optimise the parallel access to real-time and historical data. Further emphasis is put on the configurability of the system to incorporate a mission model, hereby offering a higher level of abstraction than that traditionally provided by telemetry parameters and telecommands. A new Human-Computer Interaction (HCI) concept has been adopted based on closer data integration and referential capabilities.

Concepts

SCOSII is a generic system which is configured by adding *missing specific knowledge*, which may be categorised into:

- *declarative knowledge*, e.g. calibration curves, parameter structures, etc.; specified through dedicated form based HCIs;
- *expressive knowledge*, e.g. derived parameters, command validation conditions, etc.; specified through the OL;
- *procedural knowledge*, e.g. operation procedures, report procedures, etc.; specified through the OL;
- *special knowledge*, i.e. non-generic mission information typically requiring a software expansion to SCOSII.

It is difficult to define the *borderline* of when to use *declarative* or *expressive* knowledge, i.e. when to use the OL. The definition of specific items within the database have typically *both* a declarative and an expressive part. The identifier, description, etc. of a Parameter is defined by declarative knowledge, whereas its validity criteria is defined by expressive knowledge. Due to this 'mixture' of declarative and expressive knowledge inherent to most database parts, the way the user interacts with the system needs to reflect this fact. Neither a pure (traditional) forms interface nor a pure OL definition environment would suffice, both need to be accessible in a homogeneous manner from within the same HCI.

An operations language needs to interact with the control system to be able to *access data* held by the control system which is of operational importance to get e.g. the validity status of a telemetry parameter; *request services* to e.g. send a telecommand; and *change data* to e.g. store the results on an evaluation of a derived parameter.

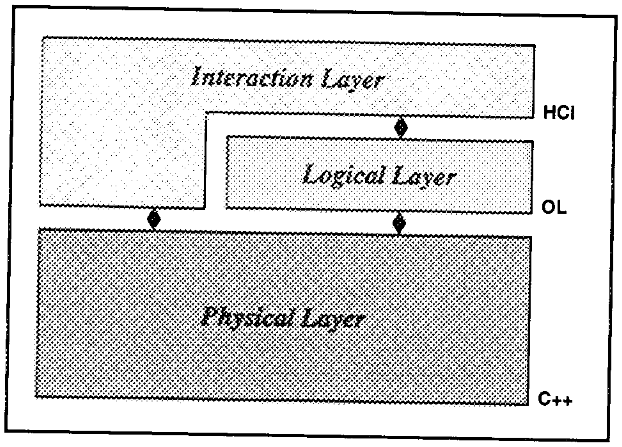


Figure-1 Layered Model

A *layered approach* has been adopted for the SCOSII OL as shown in Figure-1. The three layers are:

- *Interaction layer*, i.e. the user interface of the system which may interact with the physical layer directly or with the logical layer;
- *Logical layer*, centred around the OL containing the data entities which are manipulated via constructs in the language;
- *Physical layer*, providing the generic services of the control system.

The access from the logical to the physical layer is dictated by an explicit *data model*. The data model is *object-oriented* as it represents physical layer objects with attributes and services accessible to the OL.

It supports the explicit representation of inheritance, aggregation and association relations. This enables the OL to facilitate navigation through related objects, e.g. from a command to the parameter used within its post-execution verification checks.

The data model serves as a 'contract' between the logical and physical layers, it can not be changed through the OL itself. This does not imply that the data model is *static*, changes are just controlled through a mechanism within the physical layer. Any change to the data model is propagated to the logical layer.

The physical layer within SCOSII is itself based on an object-oriented implementation, i.e. the differences in representation between the logical and physical layers are less than would otherwise have been the case. The direct implication of this is that the logical layer is 'slim': it mainly serves to present physical layer objects to the operations engineer while hiding implementation details and offering protection against illegal access. The intelligent behaviour always rests within objects of the physical layer, i.e. if the physical layer does not support a certain function it will neither be available within the OL.

SCOSII supports the representation of a *mission model*, allowing to organise the mission knowledge according to a structural representation of system elements, e.g. a gyro or a heater. The OL can access these higher level objects in the same way as any other object within the physical layer, i.e. it does not require a language expansion to take advantage of these.

It is transparent to the OL whether it accesses *static* (database configuration data, e.g. parameter characteristics) or *dynamic* (processing data, e.g. latest parameter value) data. Although the OL does offer facilities to explicitly request *historical data*; the concept of *time* is nominally managed through the application using the OL. A parameter display may be put into *retrieval mode*, the validity of each parameter is calculated on the basis of *current values* of any contributing parameters.

It is further transparent to the OL that SCOSII is a distributed system. All aspects dealing with data distribution and synchronisation are handled fully by the physical layer.

The OL is an *interpreted language*. The reasons for this choice have mainly been that at least operation procedures are interactive of nature involving communication with a human operator - for which an interpretation was believed most adequate.

All OL definitions form part of the database configuration of a SCOSII system. They are therefore underlying strict version and configuration control.

Language Facilities

The OL is a *strongly typed* language, which enables the detection of a range of errors at preparation time during database configuration rather than causing an error at execution time. The data model forms part of the type system within the OL; accessing the physical layer objects in a wrong way will be detected prior to its execution.

The executable unit within the OL environment is an *OL Script*. A script may be as simple as a single boolean expression or as complex as the full directives of a large flight operations procedure. A script is composed of two parts: a *declaration part* (local variables and function definitions) and an *executable part* (statement list).

The access to the physical layer objects is governed through the explicit existence of an object-oriented data model. Figure-2 illustrates a segment of a script to calculate the value of the derived Parameter P117. If the status of the limit of Parameter P112 is above limits, then the engineering Value of P117 is set to the upper limit definition of P112; otherwise it is set to be the engineering Value of P112.

```
{
if (P112.limit == ABOVE_LIMITS) then
    P117 := P112.limit.upper;
else
    P117 := P112;
endif;
}
```

Figure-2 Operations Language Example

Figure-3 shows the data model corresponding to this example. A *Parameter* is characterised by its *name*, *description*, *limit*, *raw* and *engineering* Values. Each Class may have a *default attribute* (marked with a '*'): for the Parameter the default is its engineering Value. A Parameter offers a service *delta* which allows to access historical samples. A *Value* is characterised by its *value* (default) and *validity*. A *Limit* is characterised by its *status* (default), *lower* and *upper* limit definitions. Notice that due to the concept of default attributes, the expression 'P112' evaluates as 'P112.eng.value'.

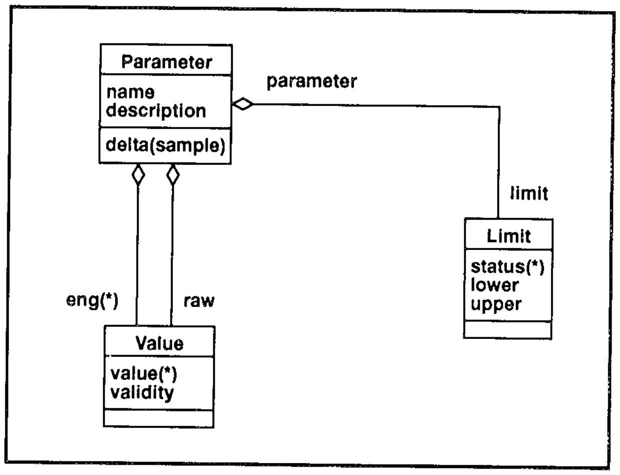
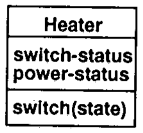


Figure-3 Logical Layer Data Model

Figure-4 shows the representation of a *Heater* system element within the logical layer. A Heater is characterised by its *switch-status* (on-off) and *power-status* (on-off) attributes, and offered service to *switch* it either on or off. The OL can operate on heaters in the same manner as on parameters shown earlier.



```
...
if (....) then
    heater13.switch(ON);
endif;
...
```

Figure-4 System Element Logical Layer Data Model and OL Example

The OL is, besides from its integration with the data model, a straight-forward imperative language. Table-1 provides an overview of the major language constructs.

Table-1 Operations Language Constructs

Statements	*Expressions*	*Functions*
assignment	value	mathematical
wait	reference	statistical
function invocation	function invocation	bit manipulation
goto-label	boolean expression	time
if-then-else	numeric expression	object creation
select-case	string expression	object copy
while-do	time expression	
repeat-until	list expression	
for-in-list -do	set expression	
for-to-step-next	matrix expression	
	vector expression	
	map expression	

The generalised approach of interfacing physical layer objects governed by the data model is not in all cases adequate. A trade-off has to be made whether to provide a more targeted syntax for particular kinds of knowledge. It is expected that specialised 'mini languages' extending the OL syntax will evolve - typically also offering dedicated HCI support. However, the baseline is that these shall be *mapped* onto the kernel OL at the syntactical level, i.e. in terms of macro expansion. This ensures that the intelligent behaviour stays within the physical layer of the MCS.

Implementation

The OL facility is implemented as any other SCOSII software: it is specified and designed using an object-oriented method (OMT, ref. [11]), and programmed in C++. The UNIX utilities lex (scanner generator) and yacc (parser generator) are used to construct the parse tree.

Due to the fact that the OL scripts form part of the database configuration and hence are defined in the preparation phase, the parse tree is built already at this stage to improve the performance in the execution phase. The parse tree structure is used directly by the interpreter.

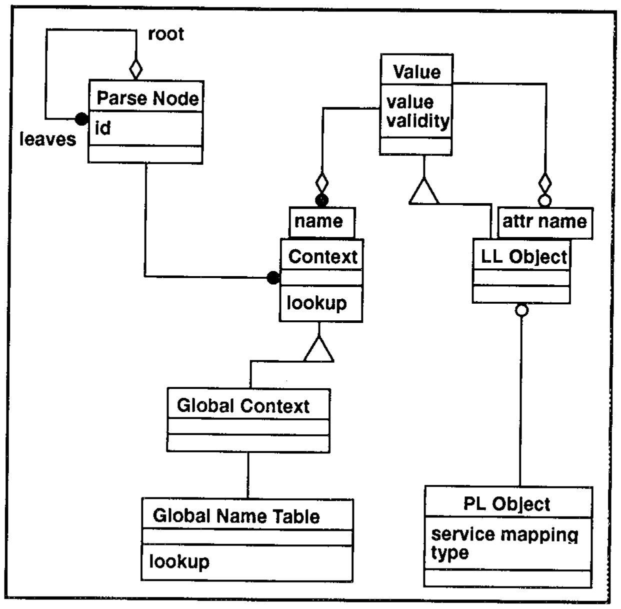


Figure-5 Physical Layer Interface Class Diagram

The *physical layer interface* is illustrated in Figure-5. A *Parse Node* is a component of the parse tree and is characterised by an *identifier.* It references its *root* Parse Node and all of its *sub* Parse Nodes. A Parse Node is evaluated within a particular Context. A *Context* maps identifiers onto Values and offers a *lookup* service. The *Global Context* is a special kind of Context which interfaces a Global Name Table provided by the Physical Layer (PL). The *Global Name Table* offers a *lookup* service taking as input a character string (e.g. “P112”) and returning a reference to the corresponding PL Object.

A *Value* is characterised by its *value* and *validity status*, which is used to propagate the effects of non-valid values throughout the evaluation of expressions: if a Value is computed on behalf of non-valid Values, it is itself to be considered non-valid.

A typical example of a non-valid Value is the state of a switched-off (or redundant) unit which still is being sampled and echoed through telemetry.

A *LL Object* is a special kind of Value. It is structured as a record, containing a Value for each of its attributes.

Any object within the PL which needs access from the LL inherits the properties of the PL Object, hereby ensuring the proper interface to the LL. A *PL Object* is characterised by its *type* and contains a *service mapping* relating requests from the LL onto C++ functions of the PL. All LL Objects are attached to one PL Object. At run-time only the PL Objects actually used are related to LL Objects.

An initiative is currently being undertaken to further generalise the physical layer interface by adopting the Model-View-Controller (MVC) architecture, ref. [15], with the purpose of using identical interfaces from both the interaction and the logical layers to the physical layer, see Figure-1. The first prototypes with this architecture have demonstrated promising results.

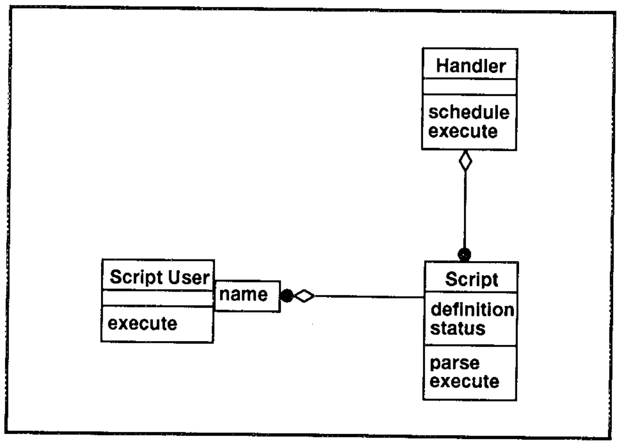


Figure-6 Script Class Model

The Handler, illustrated in Figure-6, controls the execution of any Script. It offers two services: *schedule*, which determines the order in which scripts are executed, and *execute*, which invokes the script execution.

A Script is characterised by its *definition*, i.e. a textual representation of the script, and its *status* - e.g. whether it has been parsed. It offers two services: *parse*, which builds the parse tree of the script, and *execute*, which requests the execution of the script. Any application using scripts have to inherit from the Script User class, which provides the mechanism to interface the OL environment and request the execution of scripts.

The initiative to execute scripts nominally comes from an application using the OL. The Handler has to deal with the incoming execution requests. Currently a very simple scheduling mechanism is implemented; it is foreseen to expand this into a finer-grained mechanism taking aspects, like priorities and pre-emptive scheduling, into account.

Nominally a script will be version controlled as part of its using entity: e.g. the validity criteria of a parameter specified as an OL boolean expression is seen as part of the corresponding parameter version. If the validity criteria is changed, then a new version is associated with the whole of the parameter it belongs to. The granularity in terms of at which level of detail to manage versions is decided on a mission specific basis.

No language constructs to deal with parallelism or script execution synchronisation are provided. It is believed that such aspects are better managed by the physical layer. Within the OL *conditions* can be defined as e.g. an interlock (execute upon successful verification) between two operation procedure execution requests. The physical layer knows about the conditions and observes these while servicing the related execution requests.

At this stage only basic OL editors and execution displays are provided. It is expected to expand the tools with a debugger and test tool, enabling the operations engineer to test and validate Scripts locally on a workstation.

Status

SCOSII is under development. A Basic System has recently been delivered comprising functions equivalent to those offered in the existing generic MCS's used at ESOC. A reduced OL facility covers only expressive knowledge and simple tools. Further evolutionary releases are planned:

- *release 1 (1Q95)*, adds e.g. mission modelling capabilities and executable operation procedures. The OL facility covers procedural knowledge and simple tools.
- *release 2 (1Q96)*, adds e.g. advanced mission modelling and semi-automatic operation procedure execution. The OL facility is complete with tools.
- *release 3 (1Q97)*, adds e.g. integration with knowledge based applications for automatic operation procedures execution.

FOPGEN, a WYSIWYG tool to support editing, display and printout of operational documentation, will be fully integrated with SCOSII. It provides advanced editing features and read/write access to the SCOSII mission database. FOPGEN will generate operation procedures in the SCOSII OL.

In parallel with the SCOSII development, two major studies have been initiated: *ATOS-4* exploits the use of knowledge based technology in e.g. the context of procedure execution based on SCOSII and the OL; *Productline for Compact Ground Facilities* investigates the integration of check-out and operation control systems, with particular emphasis on the language aspects.

The Committee for Operations and EGSE Standardisation (COES) is currently active to standardise the ground segment infrastructure systems within ESA. A particular subject covers the standardisation of the human-computer interaction of which a dedicated language is seen as an integral part.

The SCOSII OL will be a significant contributor to this standardisation work; the OL itself will be made compliant to the forthcoming standard.

Conclusions

The SCOSII OL provides support to the operations engineer for the configuration of a MCS with mission specific data to include expressive and procedural knowledge, hereby clarifying the borderline between the mission specific and generic elements of a MCS. The turn-around time for a change is drastically reduced as it does not involve any software modifications.

It does not cover the declarative knowledge for which the existing forms based HCI have proven to be efficient. A mixed approach has hence been adopted where only a subset of the configuration data is specified through the OL.

The existence of an explicit object-oriented data model ensures a clear framework for the interface to the physical layer of SCOSII.

The language is on purpose 'kept simple and stupid', expecting the intelligent behaviour to be provided by the physical layer objects. This facilitates improved performance within the OL environment.

The language is bound to SCOSII. As there is no intelligent behaviour within the logical layer, it depends upon the level of services offered by the physical layer. The direct implication of this is that although the architecture concepts could be adopted, it makes little sense to port the language environment to a different platform than SCOSII.

The data model approach, although flexible, has the possible disadvantage that porting OL scripts between missions can be difficult as each mission could have their own different data model. This is however a property of any generic system, not just the SCOSII OL environment.

With the planned expansions of SCOSII to cover extensive mission modelling capabilities, the added level of abstraction within the physical layer will allow the OL to take immediate advantages of this due to the generalised data model approach, without requiring syntactic nor semantic changes to the language. It is expected that the full advantages of the SCOSII OL will be demonstrated at that stage.

References

[1] SCOSII: ESA's New Generation of Mission Control Systems - The User's Perspective, P Kaufeler, M Pecchioli, I Shurmer, ESOC - *these proceedings.*

[2] A New Communication Protocol Family for a Distributed Spacecraft Control System, A Baldi, M Pace, ESOC - *these proceedings.*

[3] SCOSII: ESA's New Generation of Control Systems, M Jones, N Head, K Keyte, P Howard, S Lynenskjold - *these proceedings.*

[4] SCOSII - An Object Oriented Software Development Approach, M Symonds, S Lynenskjold, C Müller - *these proceedings.*

[5] SCOSII User Requirements Document, ESOC DOPS-SYS-URD-001-AMD, Issue 3, February 1994.

[6] SCOSII Software Requirements Document, ESOC SCOSII-SYS-SRD, Issue 0.6, June 1994.

[7] User Language Study, VEGA ULS.RST.REP.005 (ESOC), Issue 1.0, October 1992.

[8] Object-Oriented User Language for Satellite Check-out, Spacebel Informatique (ESTEC), 1992.

[9] Control File Language for Cluster AIT, CRI CL-CRI-ID-0006 (ESTEC), Issue 3, February 1993.

[10] User Reference Guide for ERS-1 Check-out Activities, CRI ER-MA-CRI-AV-065, Volume 1, Issue 1, April 1987.

[11] Object-Oriented Modelling and Design, Rumbaugh et.al., Prentice-Hall 1991.

[12] Expert Systems for Automated Spacecraft Operations, CRI/Matra EOA-CRI-FINAL-0001-1991 (ESOC), March 1993.

[13] Definition Study for a Meteosat Workstation, VEGA MWS.RST.REP.002 (ESOC), March 1989.

[14] SCOSII - A Distributed Architecture for Ground System Control, Vitrociset Keyte, International Symposium on Spacecraft Ground Control and Flight Dynamics, Brazil, February 1994.

[15] A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System, ParcPlace Systems, August 1988.