A FORMAL DEFINITION OF COSMIC-FFP FOR AUTOMATED MEASUREMENT OF ROOM SPECIFICATIONS

H. Diab, M. Frappier, and R. St-Denis
Département de mathématiques et d'informatique, Université de Sherbrooke,
Sherbrooke, Québec, Canada, J1K 2R1,
e-mail: {Hassan.Diab, Marc.Frappier, Richard.St-Denis}@dmi.usherb.ca

Abstract

We propose a formalization of the COSMIC Full Function Point (COSMIC-FFP) measure for the Real-time Object Oriented Modeling (ROOM) language. COSMIC-FFP is a measure of functional size. It has been proposed by the COSMIC group as an adaptation of the function point measure for real-time systems. The COSMIC-FFP definition is general and can be applied to any specification language. We propose a formalization of this definition for the ROOM language. ROOM is now widely used for constructing real-time systems. The benefits of our formalization are twofold. First it eliminates measurement variance, because the COSMIC-FFP informal definition is subject to interpretation by COSMIC-FFP raters, which may lead to different measurements for the same specification, depending on the interpretation made by each rater. Second it allows the automation of COSMIC-FFP measurement for ROOM specifications, which reduces measurement costs. Finally, the formal definition of COSMIC-FFP can provide a clear and unambiguous characterization of COSMIC-FFP concepts which is helpful for COSMIC-FFP measurement for other object-oriented notations like UML.

INTRODUCTION

Software development activities must be managed in order to effectively accomplish them. A high risk and cost is associated with these activities. The management of aspects like effort, quality, productivity, benchmarking, and outsourcing depends, to a large extent, on the availability of an appropriate *software size measure*. Size is a key factor to estimate effort, to compute productivity and quality ratio. Benchmarking relies on the ability to compare systems of similar size. The cost of outsourcing contracts for maintenance and evolution is also driven by the size of the software. Hence, measuring software size is an important goal for any organisation developing or using software.

Function Points is one of the prominent methods which has gained a considerable popularity in the software industry. It was proposed by Albrecht [2] and improved by *International Function Points Users Group* (IFPUG) [5]. Function points was designed to measure information systems (data intensive and transaction-based systems). Although it is conceptually applicable to real-time systems, several reports [4, 6, 7, 9] advocate that it is not very well adapted for real-time systems. Recently, function points were extended by the *Common Software Measurement International Consortium* (COSMIC) group to more adequately address real-time software. The resulting measure is called COSMIC *Full Function Points* (COSMIC-FFP) [1].

The COSMIC-FFP measure may be used to estimate development effort, evaluate software quality, manage outsourcing contracts, and compare systems, specified in different languages, in terms of productivity, quality, and maintenance costs. Consequently, these measurement aspects can be used as the basis to improve and achieve the appropriate Capability Maturity Model (CMM) level of an organization.

In [1], a measurement procedure is defined to be applied to any language specification. Since software requirement specifications depend on the language in use, the drawback of the general applicability of COSMIC-FFP is the lack of a direct representation of some specification concepts in the context of COSMIC-FFP, some object-oriented concepts for instance. This lack makes the measurement rules subject to interpretation by COSMIC-FFP raters, which may introduce some variances on COSMIC-FFP measurement when different persons apply the measurement rules to the same software specification. These variances lead to software sizing inaccuracy. Consequently, the management and control of software processes will be negatively affected. Therefore, mapping rules between COSMIC-FFP and specification concepts are required. Furthermore, the measurement rules are given in plain natural language, which makes them difficult to apply in a consistent

manner and to use as the basis for automated measurement. COSMIC-FFP measurement is a manual process; there is no tool for identifying the COSMIC-FFP elements directly from software requirement specifications.

In this paper we propose a formal definition of the COSMIC-FFP measurement procedure given in [1] for mapping COSMIC-FFP concepts to ROOM concepts. This formal definition should eliminate the variance in COSMIC-FFP measurement for ROOM specifications, because this definition is stated in mathematical terms based on the syntactic structure of a ROOM specification, hence it is objective and not subject to interpretation by the COSMIC-FFP rater. It also allows the automation of COSMIC-FFP measurement for ROOM specifications, which reduces costs of COSMIC-FFP measurement and avoids measurement errors. To the extent of our knowledge there is only one approach that has been published on measuring COSMIC-FFP for a notation (UML) [3] and it does not address the issues of formalization or automated measurement.

The ROOM language is widely used for developing real-time and distributed software. Rational Corporation provides a case tool (Rational Rose RealTime powered by ObjecTime) that allows the edition, validation, and simulation of ROOM specifications. ROOM is representative of the family of object-oriented methods for real-time systems. The ROOM language has a formal framework to abstractly and concretely describe the software. It integrates concepts of data encapsulation, inheritance, and information hiding, incorporates detail level programming language, and provides an executable model at high levels of abstraction.

The structure of a ROOM model facilitates the automation of COSMIC-FFP measurement. The ROOM notation provides the details which are essential for COSMIC-FFP measurement. In ROOM, the behavior of an object is defined as a state machine. Each machine is composed of a set of transitions between states. The structure of a transition allows to easily identify and distinguish the different types of data movements in the COSMIC-FFP context, since a transition is already decomposed into actions that are written in a programming language. The use of executable instructions in the body of actions facilitates the identification of those data movements and distinction between them, which is necessary for COSMIC-FFP measurement. We cannot easily identify these details in a specification written in semi-formal specification languages like UML.

The rest of this paper is structured in seven sections. The first section provides an overview of the COSMIC-FFP method and the ROOM language. The adequacy of different specification languages is briefly discussed in the second section. The third section formalizes some elementary concepts of COSMIC-FFP in the ROOM context which are then used in the fourth section to classify data components. The fifth section briefly describes how components are weighed. The sixth section introduces the algorithm of the COSMIC-FFP measurement for ROOM specifications. Finally, we conclude with an appraisal of this work in the seventh section, identifying its strengths, weaknesses, limits, and future work.

BACKGROUND

COSMIC Full Functions Points (COSMIC-FFP)

The following equation describes how COSMIC-FFP, denoted by $Size_{CFSU}(layer_i)$, are measured:

$$Size_{CFSU}(layer_i) \triangleq \Sigma\, size(entries) + \Sigma\, size(exits) + \Sigma\, size(reads) + \Sigma\, size(writes)$$

Variable $Size_{CFSU}(layer_i)$ denotes the functional size of the software in $layer_i$. A layer is defined as the result of the functional partitioning of the software environment. To obtain that functional size, a set of rules and procedures must be applied. They consist of mapping the software to measure onto an implicit software model and then measuring the components of this software model. There are two main categories of components: *data groups* and *data movements*. A data group is defined as a set of *data attributes* that are logically related based on the functional perspective. Data movements are divided in four categories: *entry, exit, read*, and *write*. A data movement is defined as a function which refers to a set of data attributes. The COSMIC-FFP measurement standard, 1 Cosmic Functional Size Unit (CFSU), is defined as one elementary data movement (data entry, exit, read, or write).

Sub-expressions $\Sigma size(entries)$, $\Sigma size(exits)$, $\Sigma size(reads)$, and $\Sigma size(writes)$ denote the number of all the "entries", "exits", "reads", and "writes" that are identified in $layer_i$, respectively.

The ROOM Language

ROOM is a formal modeling language for the development of systems [8]. It is well adapted to real-time systems. The design notation is a combination of graphical and textual specifications. This includes embedded segments of executable code of an object-oriented programming language.

In a ROOM model, the design might be observed via two different view points: structure and behavior. The structure represents the architecture of the model components and the links between these components. The behavior shows how the system may evolve over time. It is affected by the time and the occurrence of some events. The events are generated by the system or by its environment.

A ROOM model is based on three kinds of entity: *actors, protocols*, and *data objects*. A complete model is a combination of these entities. An actor is defined as an active object. Indeed, an object exhibits both static and dynamic semantics. The ROOM language integrates the data encapsulation concept. An actor is encapsulated and has restricted visibility of and by other actors. The access relation between the objects is by reference only. An actor offers its services that might be required by the *communication channels*. Each request of a service corresponds to a specific *message*. A protocol represents a set of messages that can be exchanged between the actors. Finally, a data object is the basic unit of the system data. It is sent or received in conjunction with the message. It is also used to define actor variables.

The dynamic part of a model is defined in the behavior section of the specification. The behavior of an actor is described as an *extended finite state machine*. In the ROOM environment, the state machines are called ROOMcharts. They are based on the Statecharts formalism of Harel. A ROOMchart is a hierarchical state machine. A state may be decomposed in sub-states. States that are not decomposed are called *leaf* states. A ROOMchart is defined by a *state context*. This state context may have variables, entry code, exit code, a set of sub-states, or a set of transitions. Executable state machines are a feature of the ROOM language.

There is a toolset that supports ROOM: Rational Rose RealTime. It was developed by Rational and ObjecTime Limited. This toolset allows the creation, modification, simulation, and implementation of a model. This includes the ability to execute an incomplete model. Rational Rose Real-time provides model editors, a set of navigators, an incremental model compiler, static and dynamic checkers, and generators of executable code in a programming language.

ADEQUACY OF THE ROOM LANGUAGE FOR FORMALIZING COSMIC-FFP

The main goal of the COSMIC-FFP method is to measure the functional size of real-time software. To measure the number of COSMIC-FFPs, a software specification document must contain the necessary details to identify and categorize:

- software features and
- data components that might be entered, modified, referred, and produced during the processing of a feature.

These details are required to determine the type of each data movement (entry, exit, read, and write) in the COSMIC-FFP context. Therefore, any evaluation of the adequacy of a specification language for the formalization of COSMIC-FFP measurement rules must take account these aspects. In the rest of this section, we evaluate briefly the adequacy of several families of specification languages for the formalization of the full function point calculation.

Trace-based specifications are not adequate for the formalization of COSMIC-FFP. Because they abstract from states, it is not possible to identify data components. Consequently, the calculation of weights for data movements cannot be fully formalized, because data components are required.

Algebraic specifications are not adequate for formalizing COSMIC-FFP definition. Operations are not always *functional processes* in COSMIC-FFP. Some of them may be auxiliary operations used in the definition of operations required by the user. We do not see any systematic way of distinguishing them. There is a similar problem with data components. Sorts may represent data components and data attributes, without any systematic way of distinguishing between them.

Because process algebras do not distinguish between inputs and outputs, it seems difficult to identify data movement components. There is a similar problem with data components. In

process algebras, event parameters may either be input parameters, output parameters or state information. Hence, it is difficult to identify data components. Because of these difficulties, process algebra are not adequate for formalizing COSMIC-FFP.

There is a more natural mapping between COSMIC-FFP and model-based specifications. However, not all model-based langages are appropriate for COSMIC-FFP formalization. For instance, in semi-formal object-oriented notations like UML and OMT it is difficult to formalize COSMIC-FFP. The main difficulty with these notations is that they are not precise enough to allow a proper classification of data movements. The necessary details for identifying data components that might be modified and/or referred are provided in plain natural language, which is almost impossible to formally analyze. For instance, in UML it is not always possible to formally determine the type of data movements or identify the data groups that are modified and referenced by the data movements based on the details provided by the sequence, class, and collaboration diagrams. In the Z notation, it is not trivial to distinguish between a modified variable and a referred variable that are used in a predicate.

In ROOM, the structure of a transition allows to easily identify and distinguish the different types of data movements, since elementary actions of a transition are written in a programming language. Data group can also be identified from data class and protocol definition. Several real-time software projects are successfully developed by a number of industrial organizations using the ROOM language. Furthermore, the ROOM language is sufficiently abstract to support good specification practices. It allows to easily construct an executable model that can be used for early prototyping and is supported by the Rational Rose toolset. Based on these observations, we have chosen the ROOM language for the formalization of COSMIC-FFP.

DEFINITION OF COSMIC-FFP CONCEPTS

In order to give a formal definition of the measurement rules, we need to first define some elementary concepts of the COSMIC-FFP measure as defined in the COSMIC-FFP Measurement Manual (COSMIC-FFP MM) [1]. We map these concepts to the ROOM notation.

Boundary and Layer

Boundary

"A boundary of a piece of software is the conceptual frontier between this piece and the environment in which it operates, as it is perceived externally from the perspective of its users. The boundary allows the measurer to distinguish, without ambiguity, what is included inside the measured software from what is part of the measured software's operating environment" [1].

Given a set of actors in a ROOM model, it is possible to derive the set of data and protocol classes referenced by these actors. Therefore, the input of the measuring process consists of the text of a ROOM specification and the set of actor names to be measured. This set of actors represents the application boundary to be measured. We denote the application boundary by B.

Layer

"A layer is the result of the functional partitioning of the software environment such that all included functional processes perform at the same level of abstraction" [1]. Each layer can be measured separately in COSMIC-FFP. In ROOM, a layer serves as an abstraction of the information hiding mechanism. It is similar to the encapsulation shell of an actor with the addition of a vertical relationship to communicate with other layers. This means that several layers might be defined at the same level of abstraction. Based on these observations we can conclude that there is no direct mapping between the layer concept of ROOM and the layer concept of COSMIC-FFP, which makes it difficult to automatically identify a layer. Therefore, a human judgment is required to select the subset of actors from the application boundary B that perform at the same level of abstraction. The selected subset corresponds to a layer in COSMIC-FFP. This provides the opportunity to measure any part of the system by selecting the set of actors representing it. Of course the layer boundary is the conceptual demarcation between this subset of actors and the other pieces of the application. We denote a layer at level i by l_i and its boundary by B_{l_i}.

Functional Process and Triggering Event

Functional Process

"A functional process is a unique and ordered set of data movements (entry, exit, read, write) implementing a cohesive set of Functional User Requirements (FUR). It is triggered by an event and, once performed, must leave the software in a coherent state with respect to the triggering event" [1].

A functional process corresponds to a *transition* in the ROOM notation. A transition is triggered by an event and implemented by a cohesive set of *actions* acting on inputs, changing the system state and/or extended state variables, and/or possibly producing a result.

Typically, a transition between two states is composed of a set of functions. There are six kinds of function: *guard condition* appearing in the transition trigger, *exit action*, *choicepoint condition*, *label action* appearing in the transition label, *entry action*, and *functions* that might be called by the previous actions. In this paper we refer to each of these actions and functions as an *elementary action*.

The arrival of a message triggers the transition execution in the following order: guard condition of the trigger, exit action of the source state (also called old state), choicepoint condition, action declared in the transition label, entry action of the destination state (also called new state), and functions that might be called during the transition processing.

Formally we define a transition by a 5-tuple as follow:

$$T \triangleq \langle s_o, Input_T, F_T, Output_T, s_n \rangle \tag{1}$$

where s_o denotes the old state; $Input_T$ denotes the set of all the incoming messages appearing in the trigger of T; F_T denotes the set of all the elementary actions associated with the transition label, s_o, and s_n; $Output_T$ denotes the set of all the outgoing messages generated by the elementary actions of T and sent to outside the boundary; and s_n denotes the new state. Note that due to the limited number of pages, we omit in this paper the formal definitions of several symbols (e.g., $Input_T$, $Output_T$).

As an elementary example, we use the specification of a traffic light system as described in the Rational Rose RealTime toolset. The requirement of this system is described as follows: "a key design requirement is that both lights cannot be green at the same time. They must synchronize with each other to make sure that light A turns red before it allows light B to turn green".

For the sake of simplicity, we would like to keep our example small. Only the TrafficLight actor (called *capsule* in the Rational Rose RealTime toolset) will be considered, illustarted in fig. 1. All its substates are leaf states. The boundary only contains the TrafficLight actor. In the rest of this paper we will use this example to illustrate the application of the formal rules.

To illustrate how we identify a functional process, consider the excerpt of the TrafficLight actor specification provided below.

```
(1) State: IHavePriority
(2)  Type:  ChoicePoint
(3)  Parent State::  TOP
(4)  Condition:  return (myId > *RTDATA);
(5) Transition: messageFromOtherLight
(6)  From::  TOP:On:Junction6
(7)  To::  TOP:IHavePriority
(8)  Event Guards:
(9)   Event Guard:
(10)   Ports:  betweenLights
(11)   Signals:  id
(12)   Guard:  TRUE
(13)  Action:
(14)   timer.cancelTimer(timerId);
(15) Transition: True
(16)  From::  TOP:IHavePriority
(17)  To::  TOP:Green:Junction3
(18)  Action:
(19)   timer.informIn(RTTimespec(10,0));
```

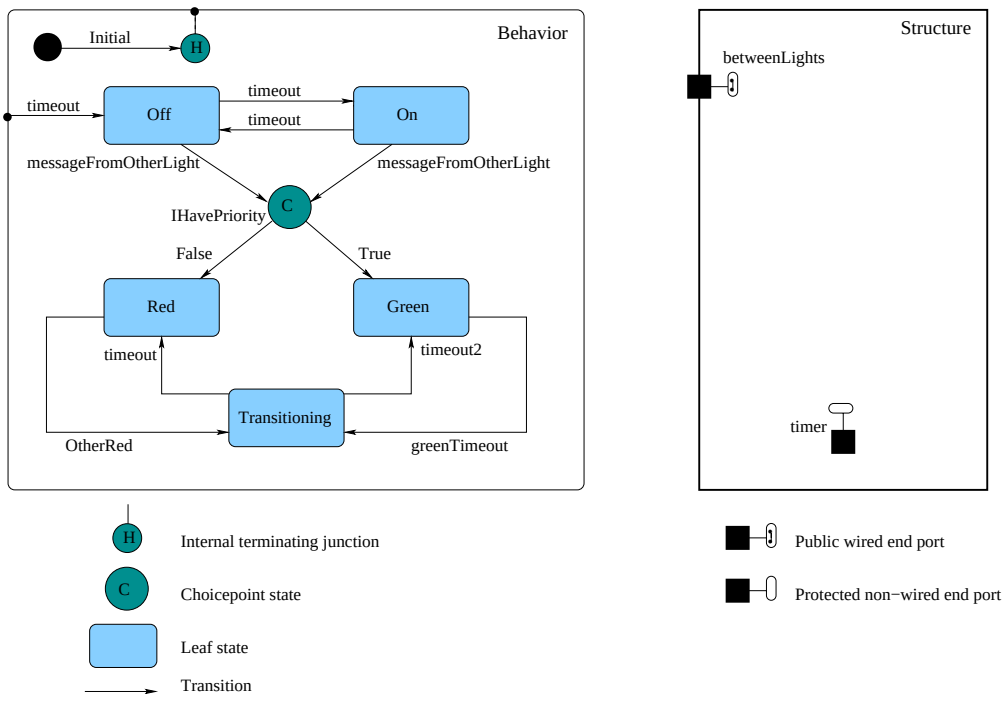

Figure 1: The behavior and structure diagrams of the TrafficLight actor

The transition between the *On* and *Green* states is composed of two segments: *messageFromOtherLight* and *True*. The transition chain execution will be triggered by the arrival of message *id* and if the choicepoint condition is true. Formally, let T_{Light} denotes this transition. Following (1), it is defined as follow:

$$T_{Light} \triangleq \langle On, id, F_{T_{Light}}, nil, Green \rangle,$$

where *On* is the source state; *id* is the incoming message; $F_{T_{Light}}$ represents the executable statements defined within the condition block of *IHavePriority* choicepoint state and the action blocks of *messageFromOtherLight* and *True*; *nil* denotes that there is no outgoing message generated during the transition processing; and *Green* is the destination state.

Triggering Event

"A triggering event occurs outside the boundary of the measured software. It may initiate one or more functional processes. Clock or timing events are considered as triggering events" [1].

The arrival of message sent from outside the boundary in ROOM corresponds to a triggering event in COSMIC-FFP, since it triggers the execution of a transition. In transition T_{Light}, the arrival of message *id* represents the triggering event of T_{Light}, since TrafficLight actor receives message *id* from outside the boundary via port *betweenLights*.

CLASSIFYING COMPONENTS

In the COSMIC-FFP definition, there are six types of component to measure: data group, data attribute, entry, exit, read, and write. In this section, we provide the informal definitions of the COSMIC-FFP measuring rules as reported in the COSMIC-FFP MM followed by our formal definitions for the ROOM notation.

Data Group

"A data group is a distinct, non-empty, non-ordered and non-redundant set of data attributes where each data attribute describes a complementary aspect of the same object of interest. A data

group is characterized by its persistence" [1]. An object in the object-oriented paradigm and an entity in the entity/relationship model are typical candidates of a data group.

In a ROOM model, actor variables and data included in messages within protocol classes represent data of the system. Actors and protocol classes refer to data classes and predefined data types in order to define the type of their components. An actor may have *simple* variables and/or *complex* variables. A variable is said to be simple if its type is a scalar data type (e.g., integer, boolean, string), an array of a scalar type, or a predefined data class. A variable is said to be complex if a non-predefined data class is used to define its type. The set of simple variables of actor a is denoted by $V_{simple}(a)$ which is defined as follow:

$$V_{simple}(a) \triangleq \{v \mid v \in variables(a) \ \wedge \ \neg(\exists\ c : c \in \mathcal{DC} \ \wedge \ type(v, c))\} \tag{2}$$

where $\mathcal{DC}$ denotes the set of all the non-predefined data classes declared in the system; $variables(a)$ denotes the set of all the variables declared in actor a; and $type(v,\ c)$ denotes that the type of variable v is data class c.

Similarly, a message within a protocol may include data. This data may be simple or complex, as actor variables. The set of simple data of message m is denoted by $D_{simple}(m)$ which is defined as follow:

$$D_{simple}(m) \triangleq \{d \mid handles(m, d) \ \wedge \ \neg(\exists\ c : c \in \mathcal{DC} \ \wedge \ type(d, c))\} \tag{3}$$

where $type(d,\ c)$ denotes that the type of data d is data class c; and $handles(m,\ d)$ denotes that message m includes data d.

A data group in COSMIC-FFP corresponds to a data class, a set of actor simple variables, or a set of simple data included within message. There are four kinds of data groups that must be identified. The following rules are given:

- Rule (4) provides the set of data classes used by the actors within the boundary.
- Rule (5) provides the set of actor simple variable groups.
- Rule (6) provides the set of data classes embedded in messages sent and/or received by the actors within the boundary.
- Rule (7) provides the set of simple data embedded in messages sent and/or received by the actors within the boundary.

$$DataClass \triangleq \{c \mid c \in \mathcal{DC} \ \wedge \ (\exists\ a, v : a \in B_{l_i} \ \wedge \ v \in variables(a) \ \wedge \ type(v, c))\} \tag{4}$$

$$DataVariables \triangleq \{V_{simple}(a) \mid a \in B_{l_i}\} \tag{5}$$

$$DataMessage \triangleq \{c \mid c \in \mathcal{DC} \ \wedge \ (\exists\ m, d : m \in M_{ext} \cup M_{int} \ \wedge \ type(d, c) \ \wedge \ handles(m, d))\} \tag{6}$$

$$SimpleDataMessage \triangleq \{D_{simple}(m) \mid m \in M_{ext} \cup M_{int}\} \tag{7}$$

where M_{ext} is the set of all the messages exchanged between the actors within the boundary and the environment via the external ports; M_{int} is the set of all the messages exchanged between the actors within the boundary.

Finally, the set of data groups, denoted by DG, used by the actors and protocols within the boundary is defined as the union of the four sets: $DataClass$, $DataActor$, $DataMessage$, and $SimpleDataMessage$.

$$DG \triangleq DataClass \cup DataVariables \cup DataMessage \cup SimpleDataMessage \tag{8}$$

To illustrate how we identify the set of data groups DG, consider the excerpt of the TrafficLight actor specification provided below.

```
(1) Capsule: TrafficLight
(2)   ...
(3)  Public Interface:
(4)   Attributes:
(5)     timerId: RTTimerId
(6)     myId: int = 0
(7)     idProvider: IdProvider
(8)   ...
(9) Protocol: BetweenLights
(10)   Type: endports
(11)   inSignals:
(12)    object Signal:  turningRed
(13)    dataType:  void
(14)    object Signal:  id
(15)    dataType:  int
(16)   outSignals:
(17)    object Signal:  turningRed
(18)    dataType:  void
(19)    object Signal:  id
(20)    dataType:  int
```

The TrafficLight actor has three attributes: *timerId, myId,* and *idProvider*. By applying rules (2), (4), and (5) to the TrafficLight actor specification, we conclude that there are two data groups: *IdProvider* defined as a data class which is used as the data type of *idProvider* attribute and the set of simple variables, denoted by V_{simple}(TrafficLight), that contains *myId* and *timerId* attributes which are typed as *integer* and *RTTimerId* (a predefined data class) respectively. To identify the data groups included in messages that are exchanged during the system processing, we apply rules (3), (6), and (7) to protocol *BetweenLights* which is used as the type of *betweenLights* port of TrafficLight actor. There is one message, *id*, that includes simple data of type integer. In this case, $D_{simple}(id)$ is identified as a data group. Finally, we apply rule (8) to identify the data groups that must be included in DG. In this example, DG contains three data groups: *IdProvider*, V_{simple}(TrafficLight), and $D_{simple}(id)$.

Data Attribute

"An attribute is the smallest parcel of information, within an identified data group, carrying a meaning from the perspective of the software's FURs" [1]. In ROOM, a data attribute is defined either as a simple actor variable or an attribute of a data class. An attribute may be simple or complex. A complex attribute may be decomposed into attributes that are defined at a lower level. For the sake of simplicity, we refer to an attribute that is not decomposed into finer attributes as an "elementary attribute". This kind of attributes represents the smallest information in the ROOM notation which corresponds to a data attribute in COSMIC-FFP. $Attributes(dg)$ denotes the set of attributes of a data group $dg \in DG$.

Data Movements

Entry

"An *entry* (E) is a movement of the data attributes found in one data group from the user's side of the software boundary to the inside the software boundary. An "entry" does not update the data it moves. Functionally, an "entry" sub-process brings data lying on the user's side of the software boundary within reach of the functional process to which it belongs. Note also that in COSMIC-FFP, an "entry" is considered to include certain associated data manipulation (validation) sub-process" [1].

In a ROOM model, an *incoming message* handles a request from outside to inside the boundary through the external ports on the interlayer boundary. A message is composed of a signal, message priority, and possibly a set of data. There are two cases defined in the COSMIC-FFP MM to identify an "entry". In the first case, if the set of data embedded in the message belongs to one data group then an incoming message corresponds to one "entry" in the COSMIC-FFP. In

the second case, if the set of data belongs to different data groups then an incoming message corresponds to several entries. We count one "entry" for each identified data group.

The number of data groups referred by a message determines the number of entries to which an incoming message must be mapped. Therefore, for each incoming external message the set of data groups that are referred in it should be identified (which corresponds to the number of entries). Note that the same principle is applied to the other data movements (read, write, and exit).

Rule (9) provides a formal definition that allows to identify the data groups included in message m within the incoming message set of transition T through the external ports.

$$Entry(T) \triangleq \{dg \mid dg \in DG \ \wedge \ \exists\, m, d : m \in Input_T \ \wedge \ m \in InMessage_{ext} \ \wedge \ (type(d, dg) \ \vee \ d \in Attributes(dg)) \ \wedge \ handles(m, d)\} \quad (9)$$

where *type*(d, dg) denotes that the type of data object d is data group dg.

By applying rule (9) to transition T_{Light}, we can identify one data group, $D_{simple}(id)$ that has been included in message id sent from outside the boundary, since message id is received and sent via port *betweenLights* which communicates with other actors (lines (10) and (11) in the specification of transition *messageFromOtherLight*).

Exit

"An *exit* (X) is a movement of the data attributes found in one data group from inside the software boundary to the user side of the software boundary. An "exit" does not read the data it moves. Functionally, an "exit" sub-process sends data lying inside the functional process to which it belongs (implicitly inside the software boundary) within reach of the user's side of the boundary. Note also that in COSMIC-FFP, an "exit" is considered to include certain associated data manipulation sub-process" [1].

Our formal definition of "exit" is similar to rule (9), except that message m must be an outgoing external message rather than incoming message. Rule (10) provides a formal definition that allows to identify the data groups included in message m generated by transition T and sent to outside the boundary.

$$Exit(T) \triangleq \{dg \mid dg \in DG \ \wedge \ \exists\, m, d : m \in Output_T \ \wedge \ m \in OutMessage_{ext} \ \wedge \ (type(d, dg) \ \vee \ d \in Attributes(dg)) \ \wedge \ handles(m, d)\} \quad (10)$$

By applying rule (10) to the transition *timeout* specification provided below, we conclude that there is no message included data that has been generated to outside the boundary during the transition processing. In this case *ExiT*(*timeout*) is an empty set.

Additional Candidates for "Exits"

In COSMIC-FFP all the external messages that do not include any data and sent to outside the boundary B during the processing of a functional process are counted as one single "exit" (e.g., confirmation and error messages). Rule (11) provides a formal definition for identifying messages without data of transition T in a single set denoted by $ExitNoData(T)$.

$$ExitNoData(T) \triangleq \{m \mid m \in Output_T \ \wedge \ empty(m)\} \quad (11)$$

where $empty(m)$ denotes that message m has no data.

By applying rule (11) to the specification of transition *timeout*, we can identify one message, *turningRed*, that will be sent to outside the boundary via *betweenLights* port when *timeout* transition is executed (line (10) in the specification of transition *timeout*). This message does not include any data. In this case *ExitNoData*(*timeout*) has one element.

```
(1) Transition: timeout
(2)  From::   TOP:Junction1
(3)  To::   TOP:off:Junction5
(4)  Event Guards:
(5)   Event Guard:
(6)    Ports:  timer
(7)    Signals:  timeout
(8)  Action:
(9)     timerId = timer.informEvery(RTTimespec(1,0);
(10)    betweenLights.turningRed().send();
```

Write

"A *write* (W) refers to data attributes found in one data group. Functionally, a "write" sub-process sends data lying inside the functional process to which it belongs to storage. Note also that in COSMIC-FFP, a "write" is considered to include certain associated data manipulation sub-process" [1].

In a ROOM model, only the elementary actions, F_T, of transition T might refer to data attributes and possibly change their values. Mapping elementary actions of a transition in ROOM to "writes" in COSMIC-FFP requires to identify the set of data attributes that are maintained by those elementary actions. F_T is said to maintain a data group dg if and only if there is an instruction in an elementary action within F_T that at least modifies the value of one attribute of dg. The group of attributes that are referred to and modified by the elementary actions within F_T will be used to determine the number of "writes" to which those elementary actions should be mapped.

Rule (12) provides the set of data groups maintained by F_T during the execution of transition T.

$$Write(T) \triangleq \{dg \mid dg \in DG \ \wedge \ \exists\, v : v \in Attributes(dg) \ \wedge \ maintains(F_T, v)\} \tag{12}$$

where $maintains(F_T, v)$ denotes that there is an executable instruction in an elementary action within F_T that changes the value of variable v, considered as an attribute of data group dg.

During the processing of transition *timeout*, there is one data group, V_{simple}(TrafficLight), that will be maintained, since *timerId*, which is an attribute of V_{simple}(TrafficLight), is maintained by an executable statement defined in the action block of transition *timeout* (line (9) in the specification of transition *timeout*). In this case $Write(timeout)$ has one data group.

Read

"A *read* (R) refers to data attributes found in one data group. Functionally, a "read" sub-process brings data from storage, within reach of the functional process to which it belongs. Note also that in COSMIC-FFP, a "read" is considered to include certain associated data manipulation sub-process" [1].

In ROOM, only the elementary actions of a transition (attributes referred without modification) may refer to a set of attributes from the storage side and bring them inside the boundary without modification. Mapping elementary actions of a transition in ROOM to "reads" in COSMIC-FFP requires to identify the set of data attributes that are consulted by F_T. F_T is said to consult a data group dg if and only if there is an instruction in an elementary action within F_T that at least uses the value of one attribute of dg.

Rule (13) provides the set of data groups that are referred by F_T during the execution of transition T.

$$Read(T) \triangleq \{dg \mid dg \in DG \ \wedge \ \exists\, v : v \in Attributes(dg) \ \wedge \ consults(F_T, v)\} \tag{13}$$

where $consults(F_T, v)$ denotes that there is an executable instruction in elementary action within F_T that refers to attribute v.

To illustrate how we identify a "read", we apply rule (13) to the specification of transition T_{Light}. During the processing of transition T_{Light}, there is one data group, V_{simple}(TrafficLight), which is consulted: attribute *myId*, which is an attribute of V_{simple}(TrafficLight), is consulted during the verification of the *IHavePriority* choicepoint condition (i.e., $return\ (myId > *RTDATA)$).

Formal Definition Summary

Table 1 presents COSMIC-FFP concepts as defined in the COSMIC-FFP MM. The first column introduces each concept with a name. The second column presents our interpretation of each concept in the ROOM notation context. For instance, a functional process in the COSMIC-FFP MM definitions corresponds to a transition in the ROOM notation.

WEIGHING COMPONENTS

In the measurement procedure given in [1], one CFSU is assigned to each data movement. The total number of CFSUs for a given layer l, which we denote by $Size_{CFSU}(l)$, is computed using the

Table 1: Mapping COSMIC-FFP concepts to ROOM concepts

COSMIC-FFP concepts	**ROOM concepts**
functional process	transition
boundary	set of actors
data group	data class or set of simple variables
attribute	elementary attribute
entry	incoming external message
exit	outgoing external message
read	consulted variables in elementary actions of a transition
write	updated variables in elementary actions of a transition

cardinality of sets $Entry(T)$, $Exit(T)$, $Read(T)$, and $Write(T)$ for each transition T in $\mathcal{T}$, where $\mathcal{T}$ denotes the set of all transitions triggered by messages sent from outside the boundary. A single data movement is counted for all outgoing dataless messages of a transition (e.g., error messages, confirmation messages). The following formula defines $Size_{CFSU}(l)$:

$$Size_{CFSU}(l) \triangleq \sum_{T\in\mathcal{T}} card(Entry(T)) + \sum_{T\in\mathcal{T}} card(Exit(T)) + \sum_{T\in\mathcal{T}} card(Read(T)) + \sum_{T\in\mathcal{T}} card(Write(T)) + \sum_{T\in\mathcal{T}} f(T) \tag{14}$$

where *card* denotes the cardinality of a set and f is a function that handles dataless messages of a transition. It is defined as follows:

$$f(T) = \begin{cases} 0 & \text{if } card(ExitNoData(T)) = 0 \\ 1 & \text{if } card(ExitNoData(T)) > 0 \end{cases}$$

THE COSMIC-FFP MEASUREMENT ALGORITHM

To compute the $Size_{CFSU}(l)$ for a ROOM specification, the information needed as input (aside from the specification itself) is the list of actors included in the boundary of layer l. This list can be provided in either of the following forms:

1. A list of composite actor names – In that case, all actor components of these composite actors are considered to be included in the boundary of layer l.
2. A list of composite actor names and a list of actors component names – In that case, only the actors specifically identified are included in the boundary of layer l.

The counting algorithm is composed of six steps, as described below. In each step, the appropriate formal rule(s), defined in previous section, should be applied.

1. Compute the set DG for all the actors identified in the input
2. Compute the set of data attributes $Attributes(dg)$ for each data group in DG
3. Determine the transitions and their components representing the behavior of the actors within the boundary of layer l
4. Compute the sets *Entry*, *Exit*, *Read*, and $Write$ of each transition
5. Compute the weight of each element of *Entry*, *Exit*, *Read*, and $Write$
6. Compute the sum of the weights to obtain $Size_{CFSU}(l)$

Of course, for the sake of concision, we have not specified in this paper how each set (e.g., DG, $Attributes(DG)$) is computed from a ROOM specification. They are rather straightforward to compute, and they shall be the subject of a technical report which will define them precisely, for the sake of completeness.

CONCLUSION

In this paper, we have proposed formal measurement rules for ROOM specifications according to COSMIC-FFP informal definitions given in [1]. The formality of our definitions constitutes a significant advantage, because it ensures that all ROOM specifications are measured in a uniform manner. This provides a greater accuracy for cost estimation, productivity evaluation and quality measurement (where COSMIC-FFP size can be used to compute defect density). It also enables a reliable benchmarking between organizations. Our formal rules of COSMIC-FFP provide a clear and unambiguous characterization of COSMIC-FFP concepts which is helpful for COSMIC-FFP measurement of other object-oriented notations like UML.

Formal rules are founded on first order logic and set theory. They could be used as the basis for developing a software size measurement tool that automates COSMIC-FFP measurement for ROOM. This tool can be integrated in the Rational Rose RealTime toolset that supports ROOM. Automation provides a good opportunity to reduce measurement costs of COSMIC-FFP and to avoid measurement errors, while making the counting process objective. Our rules are, however, specific to the ROOM notation, hence automation is directly applicable to ROOM specifications.

The formal rules reported in this paper are just the first step of our work. During the formalization process, we found several interpretations of the COSMIC-FFP rules. We have chosen the interpretation which seemed the most "reasonable", but further validations with a group of COSMIC-FFP experts will be necessary. We need to apply the formal rules to several systems and compare the results with manual measurements conducted by several experts to validate their suitability. The difference between the results will be analyzed in order to identify the elements that cause it. Implementing the formal rules in the Rational Rose RealTime toolset will streamline this validation.

In the future, we want to analyze how COSMIC-FFP could be measured from partial ROOM specifications, in order to estimate COSMIC-FFP during specification construction. We also plan to address the estimation of COSMIC-FFP for the UML notation, although UML documents do not always contain all the necessary details to measure COSMIC-FFP automatically. We are thinking of a semi-automated measurement approach where heuristics are used to identify COSMIC-FFP components from various UML diagrams.

References

[1] Abran, A., Desharnais, J. M., Oligny, S., St-Pierre, D., and Symons, C.: COSMIC-FFP Measurement Manual, version 2.0, Software Engineering Management Research Laboratory, Université du Québec à Montréal, Canada, October 1999.

[2] Albrecht, A. J. and Gaffney, J. E. Jr.: Software function, source lines of code, and development effort prediction: a software science validation, *IEEE Transactions on Software Engineering*, **SE-9** (6), July, 1983, pp. 639-648.

[3] Bévo, V., Lévesque, G., and Abran, A.: Application de la méthode Full Function Points à partir d'une spécification selon la notation UML: compte rendu des premiers essais d'application et questions. *International Workshop on Software Measurement (IWSM'99),* Lac Supérieur, September 1999.

[4] The Boeing Company: *3D Function Points Extensions, V2.0, Release 1.0, Seattle, WA: Boeing Information and Support Services, Research and Technology Software Engineering*, June, 1995.

[5] IFPUG: Function Points Counting Practices Manual, Release 4.0, International Function Points Users Group, 1994.

[6] Jones, C.: *Applied Software Measurement*, McGraw-Hill, 1991.

[7] Reifer, D. J.: Assert-R: A function point sizing tool for scientific and real-time systems, *Journal of Systems and Software*, **11** (3), March, 1990, pp. 159-171.

[8] Selic, B., Gullekson, G., and Ward, P. T.: *Real-Time Object-Oriented Modeling,* Wiley, 1994.

[9] Whitmire, S. A.: *3-D Function Points: Scientific and Reat-Time Extensions to Function Points, Proc. of the 1992 Pacific Nothwest Software Quality Conference*, Portland, OR, 1992.