Jyrki Nummenmaa, Zheying Zhang,
Timo Nummenmaa, Eleni Berki,
Jianmei Guo and Yinglin Wang

On the Generation of DisCo Specifications from Functional Requirements

DEPARTMENT OF COMPUTER SCIENCES
UNIVERSITY OF TAMPERE

D-2010-13

TAMPERE 2010

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCES
SERIES OF PUBLICATIONS D – NET PUBLICATIONS
D-2010-13, OCTOBER 2010

**Jyrki Nummenmaa, Zheying Zhang,
Timo Nummenmaa, Eleni Berki,
Jianmei Guo and Yinglin Wang**

On the Generation of DisCo Specifications from Functional Requirements

DEPARTMENT OF COMPUTER SCIENCES
FIN-33014 UNIVERSITY OF TAMPERE

ISBN 978-951-44-8284-7
ISSN 1795-4274

On the Generation of DisCo Specifications from Functional Requirements

Jyrki Nummenmaa, Zheying Zhang, Timo Nummenmaa, Eleni Berki
Department of Computer Sciences, University of Tampere, Kanslerinrinne 1, FIN-33014, Finland
{jyrki, cszhzh, cselbe, timo.nummenmaa}@cs.uta.fi

Jianmei Guo, Yinglin Wang
Department of Computer Science and Engineering, Shanghai Jiao Tong University, 800 Dong Chuan Road, Shanghai 200240, China
{guojianmei, ylwang}@sjtu.edu.cn

ABSTRACT

Requirements analysis is an important task for software development success. It is, however, often hard for various stakeholders to reach a common understanding of the behavior of the required system. In order to provide a basis for understanding the dynamic behavior of a system fulfilling the requirements, we study the possibility to automate the process of creating an executable system specification from functional requirements.

In this work, we assume that the functional requirements are formatted using a meta-model based on the classical Fillmore's case frame, which describes important semantic aspects of the documented system actions. We have constructed a method that uses grammatical conversions to produce action-based executable specifications from the requirements. This specification can be used to observe the dynamic behavior of the system, which helps to move iteration with stakeholder feedback earlier in the software development process.

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: Requirements/specifications

General Terms

Design, Verification.

Keywords

Requirements analysis, functional requirements, metamodel, executable specifications.

1. INTRODUCTION

Requirements analysis is a critical task for software development success. Typically, some kind of a high-level model is manually produced from users' textual requirements. Even though diverse techniques for requirements analysis and specification have been developed, problems still remain in this process, mainly related with the varying quality of the documented requirements and the varying expertise level of the people performing this task.

Apart from requirements analysis' tasks, it is hard to communicate with various stakeholders the outcomes of the requirements analysis phase, so that stakeholders reasonably understand the behavior of the system to be built. The late understanding of the system to be built by some stakeholders groups is a classical problem in software and information systems development, because it brings late demands and needs that need to be considered as requirements-to-be-met by the software designers.

Executable specifications provide a basis for understanding the dynamical behavior of the system to be built. With a suitable tool one can simulate and explore the behavior of the system that has been modeled. The DisCo system [1] provides tools for simulating executable specifications using a graphical user interface. The specifications in the DisCo system are based on the idea of actions, where one or more participating entities/objects participate to change the state of the system.

Our work presents an approach to generate a DisCo specification from requirements that have been preprocessed and formatted according to a meta-model, which in our case follows the classic ideas of Fillmore [7]. Our implementation is based on the user of the Meta-environment [31], utilizing high-level syntax definition and transformation rules.

The idea of working from formalized requirements specifications towards a formal representation of the desired target system is not altogether new. Probably the work closest to our approach is that of Cabral and Sampaio [5] where they generate a formal process algebra specification of the target system from system requirements, represented by using use case specification templates.

The DisCo specifications [1] are free from the complications of the concept of a process – no process design is needed in order to make the specification and to observe its behavior. This makes our method more lightweight and also more natural, as the functional requirements specifications are not based on process descriptions, either.

2. REQUIREMENTS AND THE FORMAL REPRESENTATION

Well-defined and correct requirements have traditionally been seen as a critical factor behind software project success [8, 12, 32]. Nevertheless, even if the requirements were specified correctly and precisely, it seems difficult for heterogeneous groups of stakeholders to achieve a common and correct understanding of the textual requirements, as well as the overall behavior of the system to be built. Textual requirements inherit the ambiguity of natural language, which may lead to different interpretations of the same expression. Contrary to this, a formal specification removes the ambiguity through precisely defined syntax and semantics, but it is difficult to enhance understanding of non-technical people with such a specification. In this section, we address the possibility of using formalisms for the requirements. In order to support the discussion, we start with the different types of requirements representations.

2.1 Requirements Specification

Requirements specification forms a basis for the follow-up requirements analysis and validation. It allows different kinds of representations, being more or less formal [28]. The formal representation, e.g. mathematical expression, has precise syntax and rich semantics and thus, provides a better basis for reasoning and verification, but is hardly understood by non-technical users. The semi-formal representations, such as ER diagrams, state diagrams, etc. are based on graphical modeling of the system, which provides a clear and more understandable view of the system. In contrast to semi-formal and formal presentation, an informal representation, like the requirements written in natural language, can not be used for reasoning, but its expressive power is high, and it is easy to understand. Informal presentation forms the most common way of specifying requirements, but it often inherits ambiguity from natural language, i.e. different stakeholders may understand a requirement statement differently [18,28]. This is likely to make customers dissatisfied with the implementation produced by the developers. In order to lead various stakeholders to a common understanding of the behavior of the system, instead of a careful requirements analysis and validation process, it is possible to create an executable software system specification from requirements to stimulate and explore the behavior of the required system.

The behavior of a system is commonly specified with functional requirements (FRs). FRs describe systems services or functions, and they are often expressed in terms of systems reactions to input from the environment [18]. In the textual requirements specification, due to the nature of natural language, there is no deliberate separation of an action and its associated information such as the subjects performing an action, the objects affected by an action, the instruments involved in performing an action, etc. It is tedious to parse the relevant information to automate the derivation of an executable system specification from textual FRs. The FRs shall be formulated in a way suitable for computer processing, such as a set of verb-noun pairs with the attributes connected to the verb and the noun [13]. A formal specification, with clarified semantic concerns of a FR, provides a reasonable ground to automate the generation of executable specification.

2.2 Case Grammar and its Application in Requirements Analysis

Case grammar, as proposed by Fillmore [7] is a system of linguistic analysis, focusing on the link between a verb and the grammatical context it requires. According to the case grammar theory, a sentence in its basic structure consists of a verb and one or more noun phrases, each associated with the verb in a particular case relationship, which explains various co-occurrence restrictions [7]. The coherent structure of the set of cases is a case frame, and each case represents a potential semantic slot associated with the verb. Hence, given a verb, a case frame can be defined, which consists of such semantic slots that the verb evokes. For example, the verb “submit” is necessarily associated with an objective slot (“what is submitted?”), and it may also be associated with an agentive slot (“who submits?”).

Basically, there are seven cases constituting the essential case frame [6,7]: Agentive, Instrumental, Dative, Objective, Factitive, Locative, and Comitative.

- Agentive is the concern of the agent(s) whose activities will bring about the state of affairs implied by the verb (activity). Responses to the concern are typically actors or combinations of actors found in the domain, including the system(s)-to-be, e.g. {Machine, User alone, User Supported}$_A$ choose schedule. Alternative responses to the agentive concern are essentially alternative delegations of a goal to actors (including the system-to-be).
- Instrumental is the concern that determines the instrument that is involved in the performance of the generic activity implied by the verb, e.g. Pay {by debit, by credit, by cash}$_{Ins}$.
- Dative is the concern of the agent(s) who will be affected by the generic activity implied by the verb. As above, responses to the concern are typically actors or combinations of actors found in the domain, including the system(s)-to-be, e.g. Send a message to {the admin, the user}$_D$, Notify {designated nurse, nurses at nursing station} $_D$.
- Objective is the concern of the object(s) that is affected by the generic activity implied by the verb, e.g. Send {an e-mail message, a fax message}$_O$, Print {a full report, a summary}$_O$.
- Factitive is the concern of the object(s) or being(s) that is/are resulting from the activity or understood as part of the meaning of the verb e.g. Format Text {bold, italic}$_F$ or Turn light {on, off}$_F$.
- Locative is the concern about the spatial location(s) where the generic activity that is implied by the verb is supposed to take place, e.g. Send a message {in the Car, on a Bus}$_L$.
- Comitative is a case that denotes companionship. It carries the meaning “with” or “accompanied by”, e.g., discuss the plans {with an expert}$_C$.

Considering the verb that describes the generic activity in a FR, the requirement refinement can be driven by the semantic slots associated with that verb and the corresponding elements [23]. That is to say, verbs expressing an action always express the change of an object, initiated by an agent, from one state to another, etc. Many researchers followed the way opened by Fillmore, proposing different approaches to filling in the gap between the informal requirements representation and the formal model for software development. The research is mainly conducted from two perspectives, (i) requirements specification and (ii) transformation from textual requirements to formal design models.

From the perspective of requirements specification, many researchers elaborate on the various concerns in product line engineering, and adapt the case grammar to construct variability frame for goals and FRs in product line engineering ([11, 23, 26]. The case frame provides possible types of concerns which provide a basis for understanding language semantics in a requirements engineering context.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference'10, Month 1–2, 2010, City, State, Country.
Copyright 2010 ACM 1-58113-000-0/00/0010…$10.00.

From the perspective of model transformation, the research mainly focuses on transforming the textual requirements to formal use case specifications by using the case frame, which can be further transformed to other formal models. For example, Rolland and Achour [29] proposed an approach to progressively transforming initial and partial natural language descriptions of scenarios into well structured, integrated use case specifications. Cabral and Sampaio [5] proposed a controlled natural language use case specification templates based on the case frame. They applied the templates to generate process algebraic formal models from use cases automatically.

Due to the ambiguity of natural language and the different levels of the formalism between the natural language and the formal specification, there have been very few attempts to automate the conversion from requirements to a formal specification language [22]. In our study, we propose an action-based executable specification generation from the FRs formulated by case grammar. Unlike the above cited approaches, our approach focuses on simulating the behavior of the software system in question. We assume that the textual requirements have been parsed and formatted on the basis of the predefined requirements meta-model. The transformation occurs between the preprocessed requirements and the executable specification. The specification system DisCo, used in our work, can be further used to simulate the executable specifications[1]. The simulation shows realization of the dynamic behavior of the system, and enhances the clarity and understandability of FRs

3. A METAMODEL OF FUNCTIONAL REQUIREMENTS

We use Fillmore's case frame as a basis for interpreting language semantics in FR specification. The most essential aspects can be captured in a meta-model, as shown in Figure 1. The rectangles represent the objects that compose a FR. The links represent the "property of" relationship between objects, i.e. an Action has property of Agentive, Ojective, etc. and an Agentive has property of State. The metamodel consists of ten objects, and each object has a property, Name. These objects can be grouped into three categories, i.e. the action, the case frame of the action, and the state.

An action, simply specified by a verb-noun pair, presents a primitive FR. Conducting an action results in the change of a state. Each action is associated with a number of semantic concerns, i.e. the case frame of the verb, which includes Agentive, Objective, Dative, Factitive, Instrumental, Locative, temporal, and Conditional. Some of the concerns are obligatory, such as Agentive, while the others are optional. An obligatory concern has at least one instance, associated with an Action instance. Compared with Fillmore's case frame, we removed Comitative, but add Temporal and Conditional into the concerns. A temporal concern refers to the duration or frequency of an action implied by the verb [23]. A Conditional concern refers to alternative triggers of the action or alternative conditions under which the action can be fulfilled. It is an important object to explain the operational dependency between actions.

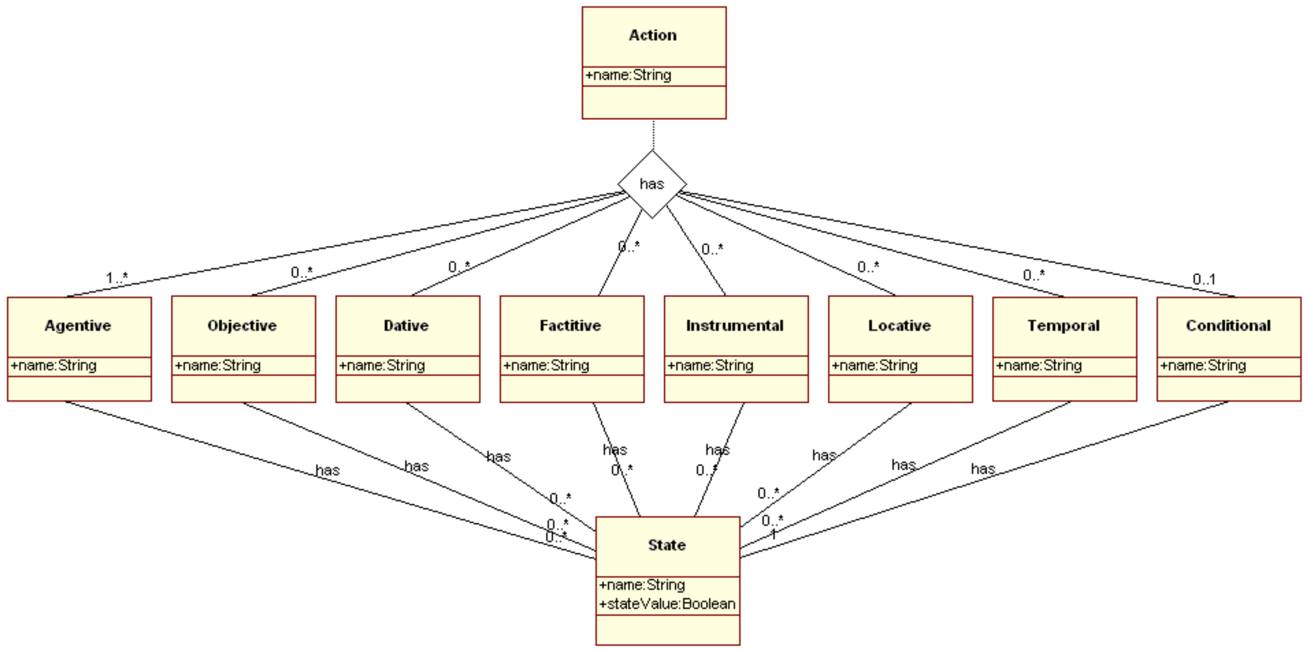


Figure 1. A metamodel of functional requirements

A state is a snapshot of the world at an incident [13]. It contains the description of the software system in question and its environment. Each object of the case frame is associated with states and their corresponding values. As shown in the metamodel, the number and value of these states will vary, depending on the action to be fulfilled. In each action, Conditional always has a state whose value determines the fulfillment of the action. A change from one state to another state is led through an action of a system. The action is triggered by the state at an incident.

The metamodel specifies every primitive FR into an action and its associated semantic concerns. This is by no means a complete list but rather a guideline for basic transformation between requirements and the action-based formal specification. The relationship between each object is simple and straightforward. There are no complex traceability links and requirements dependencies, which are hidden into the individual objects. The statement of Conditional, together with the state value of other objects, implies the dependency relationship between requirements and controls the flow of actions. An example of the FR formatted on the basis of the metamodel is illustrated in Section 5.1.

4. FORMAL SPECIFICATION METHODS IN SOFTWARE DEVELOPMENT

4.1 Formal Methods

Formal Methods (FM) is a name given to describe a particular and often neglected family of many different software development methods; all comprise mathematical specification techniques, applied though in a limited fashion and utilized in selected application domains. FM, being rigorous and abstract enough by their mathematical nature, have been used to model complex systems as mathematical objects. By building a rigorous model of a complex system, it is mainly possible to (i) handle complexity by abstraction and (ii) verify the system's properties with more systematic principles and logic than empirical testing techniques.

Aims (i) and (ii) have been major software challenges in software engineering theory and practice, with successful and unsuccessful examples of FM applications to show the benefits and drawbacks of rigorous descriptions. Using formality over unnecessarily descriptive details that many conventional methods support with time-consuming techniques is a step to improve system design and delivery times. Abstraction and formality have also been considered as promises to improve system understandability and quality [10] but cannot guarantee absolute correctness. Formal requirements specifications, though, can easily be checked by various FM support tools such as specification editors, type checkers, consistency checkers and proof checkers; these should indeed reduce the likelihood of human error.

A formal specification of a system is a mathematical abstraction of the real system. It could have a potential number of implementations in various programming languages whose syntax, semantics and grammar allow the formal logic of the real system to be expressed. However, the mathematical disciplines used to formally describe real and software systems requirements do not necessarily provide a computational model [2]. In addition, the metamodels used by most formal methods are often limited in order to enhance provability. There is a notable tradeoff between the need for rigor and the ability to model all behaviors and changes, which leads to the likelihood of errors. Certainly the use of particular FM that support dynamic systems modeling can give increased confidence in the implementability and in the integrity of the system and more confidence that the system will indeed perform as expected but errors still exist due to the dynamics of the human components.

FM vary a lot in formality and abstraction and they employ similar but also very different techniques, semantics and logic to formally model requirements. Moreover, not all formal notations capture change and system dynamics naturally, unless they are customized [30], or unless they are combined with other conventional methods [3,4].

Formal logic is also scholastic and detailed and can even be time-consuming and resource-consuming, unless very suitable tool sets and proving strategies exist and have been proved to be time-saving and cost-effective [9]. Even where full formal development is employed, for instance when the specification is refined to executable code, there might still be further metalogic and metasemantics to be employed at another level of abstraction [16], to question the self-efficacy of the specification and the use of the particular programming constructs.

4.2 DisCo Specifications

DisCo is primarily intended for the specification of reactive systems and its semantics have been defined with the Temporal Logic of Actions [20]. Among previous work, a DisCo specification has for instance been successfully created for a mobile robot case study [25]. In the study, a specification was first created that represents how the mobile robot, a small microcontroller-based car, operates. The specification was later implemented in the C language. In another case, it was used to create a specification of an on-board ozone measuring instrument, intended to be attached to an earth-orbiting spacecraft [25]. The time needed to complete the specification was 1.5 man-months, including the time it took to get to know the instrument. A version of DisCo with added probabilistic features has also been used used to research the usage of formal specifications in game design [27].

The DisCo software package, originally developed at the Tampere University of Technology, includes a compiler for compiling specifications created in the DisCo language, a graphical animation tool for animation and simulation of those specifications, and a scenario tool for representing execution traces as Message Sequence Charts [1].

The DisCo language [15] itself is a broad language and therefore we will only concentrate here on five basic parts. These parts are layers, classes, actions, relations and types. A thorough explanation of the language is given by Järvinen [14].

In execution of a DisCo specification, the state of the system is represented by objects that are instances of classes. The classes are made up of variables that can be pointers to other objects called references or other types such as integer, real, time, boolean, record type, set and sequence. An object can also be in several states which are in practise defined by enumerations. The object is always at one of the states of each enumeration.

In addition to the types provided by the DisCo system, extending types and introducing new types is possible. An example class declaration in the DisCo languages is given in Figure 2.

```
class ExampleClass is
   exampleState: (active, passive);
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference'10, Month 1–2, 2010, City, State, Country.
Copyright 2010 ACM 1-58113-000-0/00/0010…$10.00.

```
    exampleState2: (alive, dead);
    exampleInteger: integer;
    exampleReal: real;
    exampleBoolean: boolean;
    exampleTime: time;
    exampleObject: reference ClassName;
    exampleSet : set integer;
    exampleSequence : sequence integer;
end;
```

Figure 2. A class that can be instantiated as an object

Relations are logical relations between objects. Only objects can participate in relations and only binary relations are allowed. The relation can however be a partial function, total function, injection, surjection or bijection. Objects can be set to be in a certain relation or removed from the relation at runtime.

Actions alter the state of the system being executed by altering the values of the variables in objects and the contents of relations. The actions can, however, only alter the values of variables in participant objects which are specified for the action. Actions contain a guard, basically in the form of an if statement. An action is said to be enabled if the guard evaluates to True and not enabled if the guard evaluates to False. The operations to be performed when the action is run are specified. This part also supports an if/else mechanism for better control of the operations. The order of the operations separated with the || notation will not have an effect on the execution of the action. An example action is given in Figure 3.

```
action exampleAction(a: ClassA;
      b: ClassB; c: ClassC) is
when  (a.state'active  and  related(a,b)
   and c.integerValue = 3) do
    if (b.booleanValue = false)
          b.booleanValue := true;
    else
          a.state->passive() ||
          c.integerValue = 3 ||
          b.booleanValue := true ||
          not related(a,b);
    endif;
end;
```

Figure 3. An example action

Finally, layers contain all things that make up a specification. Layers may also import other layers making modular development of the specification possible. One layer may import several layers and layers that have already imported other layers.

The DisCo animator is a graphical tool for preparing the actual simulations and then executing them. The part of the animator, which determines how the execution progresses, is the execution model. With specifications that can be executed, simulated or animated, it is important to have an abstract execution model [19]. The first step in executing the specification, with this execution model, is always choosing the action to execute next. This can be automated or the action can be chosen by hand in the UI. When the selection of the action is automated, the animator chooses one of the enabled actions based on a weighted draw. It is also possible to have the animator continuously execute new actions automatically. Figure 4 shows a screenshot of the DisCo animator.

5. GENERATING DISCO SPECIFICATIONS FROM METAMODEL BASED REQUIREMENTS

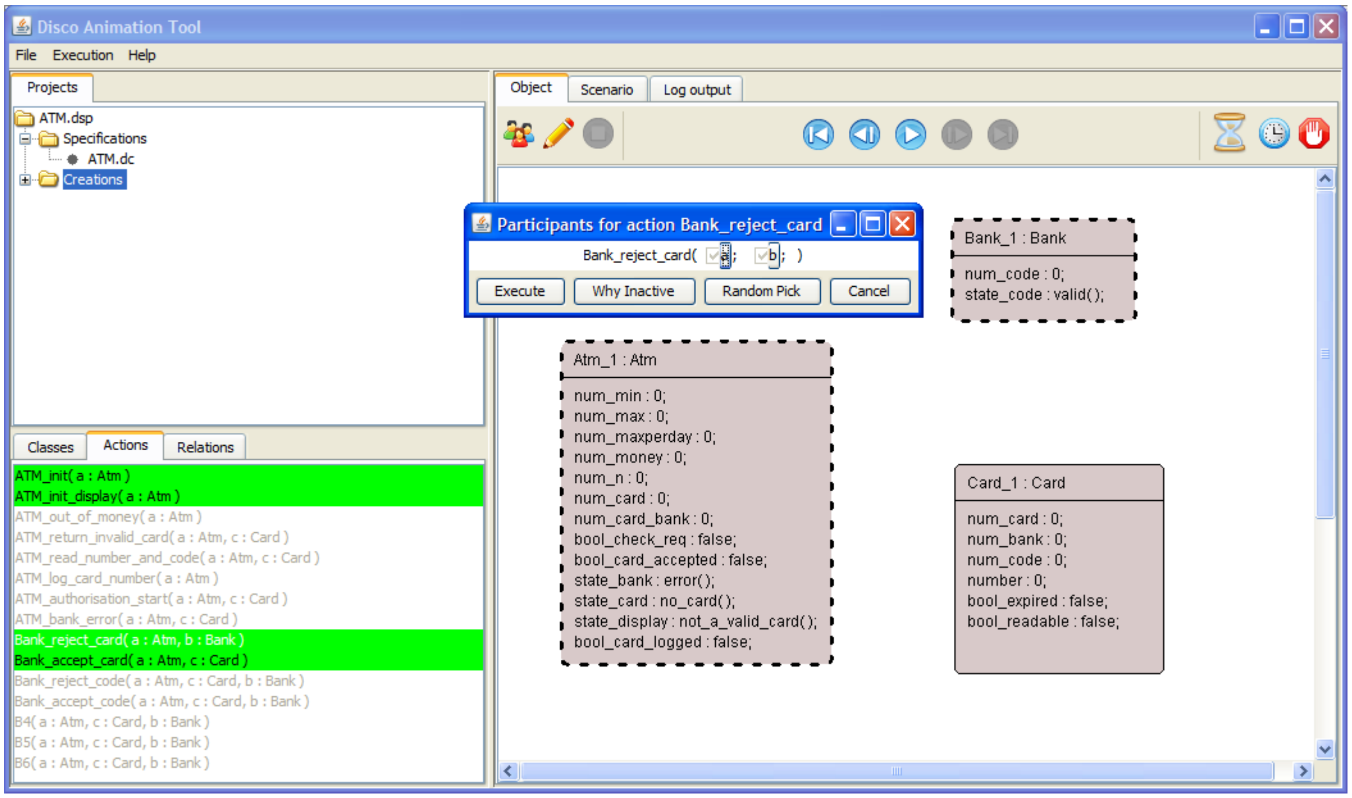
5.1 An overview of the method

Sections 3 and 4, above, indicate that the metamodel based FRs are essentially quite similar to the DisCo specification, and thereby, it is a reasonable choice to generate a DisCo executable specification to observe the behavior implied by the FRs. A single functionality corresponds well with the idea of a DisCo action. Both describe participants, a precondition, and the effect of the execution of the function or the action. They are not concerned with concepts like processes or such explicitly – the preconditions and the changes of states guide the flow of the execution in both cases.

The above observation forms the basis for the actual DisCo specification generation. Let us discuss the details using an example. Let us consider a requirement specification for Automated Teller Machines. Due to lack of space, we can not discuss the entire specification here, but just some parts of it.

Let us consider a functional requirement, which says that, if the ATM receives a negative authorization response (one of “Bad password”, “Bad bank code” or “Bad account”) from the bank information system, it will display a message to the user and eject the card.

The example requirement clearly contains a conditional (response from the bank information system), participants/agents (the bank information system and the atm, objectives (the card being ejected) and factitives/changes in the state (change of atm display status, change of atm card status).


Disco Animation Tool
File Execution Help
Projects
Object Scenario Log output
Classes Actions Relations
Participants for action Bank_reject_card
Execute Why Inactive Random Pick Cancel

ATM.dsp
Specifications
ATM.dc
Creations

Bank_reject_card(a; b;)

```
ATM_init( a : Atm )
ATM_init_display( a : Atm )
ATM_out_of_money( a : Atm )
ATM_return_invalid_card( a : Atm, c : Card )
ATM_read_number_and_code( a : Atm, c : Card )
ATM_log_card_number( a : Atm )
ATM_authorisation_start( a : Atm, c : Card )
ATM_bank_error( a : Atm, c : Card )
Bank_reject_card( a : Atm, b : Bank )
Bank_accept_card( a : Atm, c : Card )
Bank_reject_code( a : Atm, c : Card, b : Bank )
Bank_accept_code( a : Atm, c : Card, b : Bank )
B4( a : Atm, c : Card, b : Bank )
B5( a : Atm, c : Card, b : Bank )
B6( a : Atm, c : Card, b : Bank )
```

```
Bank_1 : Bank
num_code : 0;
state_code : valid();
```

```
Atm_1 : Atm
num_min : 0;
num_max : 0;
num_maxperday : 0;
num_money : 0;
num_n : 0;
num_card : 0;
num_card_bank : 0;
bool_check_req : false;
bool_card_accepted : false;
state_bank : error();
state_card : no_card();
state_display : not_a_valid_card();
bool_card_logged : false;
```

```
Card_1 : Card
num_card : 0;
num_bank : 0;
num_code : 0;
number : 0;
bool_expired : false;
bool_readable : false;
```

Figure 4. A DisCo animator screenshot

Using the metamodel presented in Section 3, the requirement could be structured as follows:

```
Agentive atm;
Agentive bank;
Objective card;
Conditional Bank response is "bad password"
or "bad bank code" or "bad account";
Factitive Atm's card status is changed to
ejected;
Factitive Atm's display status is changed to
an error message;
```

For the DisCo specification generation, though, we assume that some further preprocessing would be desired, to produce the following::

```
Agentive atm a;
Agentive bank b;
Objective card c;
Conditional b.response = bad_passord or
          b.response = bad_bank_code or
          b.response = bad_bank_account;
Factitive atm.card_status := ejected;
Factitive atm.display_status :=
           card_not_authorized;
```

The generation of a corresponding DisCo action now becomes reasonably straightforward. Each FR is used to generate a DisCo action with the following rules:

1. The Agentives, Objectives, Datives, Locatives and Instrumentals are used as objects in the action's variable declarations.
2. The Conditional forms the `when` condition of the action.
3. The Factitives are used to generate assignments of the DisCo action.

In addition to the actions, it is also necessary to generate the classes for the DisCo specification. The required information for this is spread around in the FRs, and classes are formed as follows:

1. The FRs are scanned to make up a list of classes.
2. Based on the variable definitions in each FR, for each class, look for all attributes used in the variables of that class in the factitive and conditional parts of the FRs. These attributes make up the set of attributes for the class.
3. For each state-valued attribute, search for all states used in the comparisons and assignments in the FRs, to include the states in the declaration of that attribute.

The DisCo action generated from our example FR is given below in Figure 5.

```
action ATM_return_invalid_card ( a : atm;
       c : card ) is
  when ( a.state_card'new_card )
     and ((c.bool_expired = false) or
         (c.bool_readable = false))
  do  a.state_display->not_a_valid_card() ||
      a.state_card->no_card();
  end;
```

Figure 5. A generated DisCo action

The DisCo specification generation is fairly straightforward, and, if the preparation of the FRs for the conversion has been successful, an executable specification can be automatically generated from there, and the specification can be read with the DisCo animator tool to explore the dynamic behavior of the system. The screenshot in Figure 4, above, is related to our ATM example.

5.2 On the use of the Meta-environment in the specification generation

The implementation used in this work is based on the use of the Meta-Environment [30], which is a "grammar-ware" environment, offering high-level tools for working with formal languages. The environment offers advanced grammar-ware technologies, which have benefits far beyond the needs of the present work.

This work employs the Syntax Definition Formalism (SDF) of the meta-environment. The SDF combines the lexical and the syntax definition into a single definition and it includes disambiguation mechanisms such as rejects and priorities. The translation from one language to another can be implemented using fairly straightforward rewrite rules. When rewrite rules are added to the SDF, the resulting technology is called ASF+SDF. The ASF+SDF can be used in several ways, but probably the easiest is to use the Meta-Environment, which offers integrated support for the development and testing of ASF+SDF. The latest development of the Meta-Environment is called the Rascal Meta-Environment, but that part has not been utilized in the present work.

Even though the SDF formalism is succinct, the resulting design is too long to be included here. The following code snippets, separated by "…", are examples of SDF definition of the requirements specification and the DisCo specification language definitions.

```
[A-Za-z][A-Z0-9a-z\_]* -> ReqNameStr
…
ReqNameStr -> ReqClassName
ReqNameStr -> ReqName
ReqNameStr -> ReqVarName
…
"requirement" ReqName REQOPTIONS
   "endrequirement" -> REQUIREMENT
…
{REQOPTION";" }* -> REQOPTIONS
"agentive" ReqClassName ReqVarName
   -> REQOPTION
"factitive     "ReqVarName"."ReqStateAttrName
   ":=" STATEEXP -> REQOPTION
"factitive"      ReqVarName"."ReqBoolAttrName
   ":=" BOOLEXP -> REQOPTION

"factitive" ReqVarName"."ReqNumAttrName
   ":=" NUMEXP -> REQOPTION
"dative" ReqClassName ReqVarName ->REQOPTION
"location" ReqClassName ReqVarName
   ->REQOPTION
"conditional" BOOLEXP -> REQOPTION
```

Even though the above does not give a complete syntax definition, it gives some taste of the definition style and the resulting grammar. For the translation, also the DisCo language needs to be described, re-using the definitions for names, conditionals, and such as much as possible. The re-use can be facilitated by including information from other SDF modules:

```
"action" ReqName
   "(" {DISCODECLARATION";" }* ")" "is"
   "when" BOOLEXP "do"
      {DISCOASSIGNMENT"||" }*
   "end " ReqName -> DISCOACTION
```

The above re-uses the requirement name syntax for DisCo action names. Also the Boolean expressions follow the same formalisms, and BOOLEXP is in fact defined along with the requirements syntax. The Meta-environment generates automatically a parser, which is helpful in grammar development.

The transformation from one language to another is implemented with a set of rewrite rules. Even though the rules work on a high level, space limitations prohibit extensive discussion of the transformation from FRs to DisCo actions. The following rules give exemplify the rules that are being used..

```
trp(requirement_specification
        DiscoLayerName Requirement*) =
    layer DiscoLayerName is
       tra(Requirement*)

tra(requirement DiscoActionName ReqOption*
        endrequirement; Requirement*) =
  action DiscoActionName ( trd(ReqOption*) )
      is when trc(ReqOption*)
        do trs(ReqOption*)
     end DiscoActionName;
        tra(Requirement*)
```

Above, the function `trp` is the top-level transformation function and `tra` takes care of transformation of a single action, passing the remaining list of actions recursively to itself. The functions `trd`, `trc`, and `trs` take care of extracting the required declarations, conditional, and assignment statements, respectively.

5.3 Implementation-related discussion

Even though the language used to describe the FRs is somewhat limited, it serves as a good input for an executable specifications. As we are working on a system-level description, there is no need to know or identify internal subsystems of the systems that are being specified.

In the present implementation the temporal aspects are not utilized in the specification, nor is the FR aspect Temporal. The temporal assignments and conditions in the the DisCo system are otherwise treated similarly as normal attributes are treated, but there is a global variable `now` that can be used to access the current time of the simulation. However, the simulation time in the current version of DisCo works basically on the logical level (counting ticks) and the main emphasis is on the logical order of the events, so to describe and use wall clock or calendar times, some specification of simulation level conversion would be needed.

The lack of data types in the FRs could be overcome by either completely implicit type system, where the Boolean expressions and the assignments are used to calculate the types of the attributes in classes. Even though this would be user-friendly if successful, it is also more error prone, as missing type information may prohibit the generation of a working specification. In the case of missing type information, so we chose to use explicit type information. Instead of using separate attribute declarations in the FRs, the variable names are prefixed to indicate type (e.g., variable `bool_working` is taken to be of type Boolean).

Finally, the high-level tools of the Meta-Environment used here allow for easy modification of the grammars and the transformation, and the proposed model implementation can quite flexibly be changed to adopt to a somewhat different input or to utilize some new feature of the DisCo language.

6. IMPLICATIONS TO THE SOFTWARE DEVELOPENT PROCESS

The software development processes tend to include iteration, caused by testing and stakeholders' feedback. The attempt is to move this iteration as early in the process as possible.

If requirements are formalized and specifications are generated from them, then it is possible to move stakeholder feedback and some of the resulting iteration earlier in the software development process – in fact it is possible to iterate the requirements specification, executable specification generation, animation, and stakeholder feedback cycle to take place before any attempt to design or prototype the system is even started, as follows.

1. FRs are specified.
2. Executable specifications are generated from the FRs.
3. The executable specification is animated to observe the behavior of the required system.
4. If there is a need to modify the FRs, move to Step 1.

Thus, starting to employ this kind of a methodology also implies changes to the software development process.

It has been observed that formalizing the requirements tends to increase their quality [5, 21, 22, 33]. We have only used our approach in a laboratory setting this far, but the findings are exactly the same. It is very easy to overlook details in the textual representation of the FRs. Once FRs need to be converted into a working executable specifications, details have to be filled in.

The use cases or user stories are often used to specify some example behavior of the desired system. However, typically such a system can produce many other behaviors also. The DisCo executable specifications can be used to explore, not only a specified use case or user story behavior, but also basically any type of behavior that the system produced. The resulting executable specification also works as an important description of the behavior of the system to be implemented.

7. CONCLUSIONS

In this work, we present a natural method to generate executable DisCo specifications from functional requirements. The DisCo system [1] contains an interactive animation tool, in which the user can guide the execution of the generated specification, choosing amongst the actions that are enabled for execution, given the system state. This provides an easy and straightforward way to observe the dynamic behavior of the system that has been specified. The executable specification generation does not need or use any design information on processes etc., it works on the level of the functional requirements.

The obvious complication of this method is that the requirements need to be represented and formalized according to a meta-model. For the method to be truly automated, it should work directly from textual representation, rather than a meta-model based representation.

However, if the transformation is utilized, notable benefits can be achieved: The behavior of the system can be observed from the specification and at the same time the quality of the functional requirements is improved, when they are tested with the specification.

8. REFERENCES

[1] Aaltonen, T., Katara, M. and Pitkänen, R. DisCo toolset – the new generation. Journal of Universal Computer Science, 7(1):3–18, 2001.

[2] Berki, E. Formal Metamodelling and Agile Method Engineering in MetaCASE and CAME Tool Environments. Tigka, K. & Kefalas, P. (Eds) The 1st South-East European Workshop on Formal Methods. Agile Formal Methods: Practical, Rigorous Methods for a changing world (Satellite of the 1st Balkan Conference in Informatics, 21-23 Nov 2003, Thessaloniki). Pp. 170-188. South-Eastern European Research Center (SEERC): Thessaloniki, 2004.

[3] Berki, E. & Georgiadou, E. Towards resolving Data Flow Diagramming Deficiencies by using Finite State Machines. I M Marshall, W B Samson, D G Edgar-Nevill (Eds) Proceedings of the 5th International Software Quality Conference. Universities of Abertay Dundee & Humberside, Dundee, Scotland, Jul 1996, ISBN: 1 899796 02 9.

[4] Berki, E. & Novakovic, D.. Towards an Integrated Specification Environment (ISE). Katsikas, S. (Ed.) Proceedings of the 5th International Hellenic Conference of Informatics. Athens, Greece, 7-9 Dec 1995. pp. 259-269, Greek Computer Society, EPY: Athens.

[5] Cabral, G. and Sampaio, A. Formal Specification Generation from Requirement Documents. Electronic Notes in Theoretical Computer Science, 195(18): 171-188, 2006.

[6] Cook, W. A., SJ, Case Grammar Theory. Washington, DC: Georgetown University Press, 1989.

[7] Fillmore, C. J. The Case for Case. In Bach and Harms (Ed.): Universals in Linguistic Theory. New York: Holt, Rinehart, and Winston, 1-88, 1968.

[8] Hofmann, H. F. and Lehner, F. Requirements engineering as a success factor in software projects, IEEE Software, pp. 58-66, July/August, 2001.

[9] Georgiadou, E., Siakas, K. & Berki, E. Quality Improvement through the Identification of Controllable and Uncontrollable Factors in Software Development. Messnarz, R. & Jaritz, K. (Eds) EuroSPI 2003: European Software Process Improvement, EuroSPI 2003 Proceedings, 10-12 Dec 2003, Graz, Austria. Pp. IX 31-45. Verlag der Technischen Universität: Graz.

[10] Georgiadou, E. & Berki, E. Improving Systems Specification Understandability by Using a Hybrid Approach. M Bray; H-J Kugler; M Ross; G Staples (Eds) INSPIRE I Process Improvement in Teaching and Training. First International

Conference on Software Process Improvement, Research, Education and Training. (INSPIRE '96), Sep 1996, Bilbao, Spain. Pp. 137-147, SGEC Publications

[11] Guo J. Wang Y. and Zhang Z. A Model-Driven Approach to Developing Domain Functional Requirements in a Product Line, submitted to Information and Software Technology (under review)

[12] IEEE Recommended practice for software requirements specification. IEEE Standard 830-1998, 1998.

[13] Jacobsen, K., Sigurjónsson, J., and Jakobsen, Ø. Formalized specification of functional requirements, Design Studies, 12 (4), October 1991, Pages 221-224

[14] Järvinen, H-M. The DisCo2000 Specification Language Annotated version. 2002.

[15] Järvinen, H.-M. and Kurki-Suonio, R. The DisCo language and temporal logic of actions. Technical report, Tampere University of Technology, Software Systems Laboratory, 1990.

[16] Karakitsos, G., Berki, E. & Georgiadou, E. LEARN: The Logical Entities Analytic Rule Notation, An Alternative Formal Semantics Definition. Al-Ani, B., Arabnia, H. R. & Mun, Y. (Eds) Software Engineering Research and Practice, SERP '03, Vol. II, Las Vegas, Nevada. pp. 871-876, CSREA Press, USA.

[17] Katara, M. Composing DisCo specifications using generic Real-Time events - a mobile robot case study. In J. Penjam, editor, Software Technology, Proceedings of the Fenno-Ugric Symposium FUSST'99, Technical Report CS 104/99, pages 75–86, Sagadi, Estonia, 1999. Tallinn Technical University.

[18] Kotonya, G. and Sommerville, I., Requirements Engineering - Processes and Techniques, John Wiley & Sons, 1998

[19] Kurki-Suonio, R. A Practical Theory of Reactive Systems: Incremental Modeling of Dynamic Behaviors. Springer, 2005.

[20] L. Lamport. The temporal logic of actions. ACM Trans. Program. Lang. Syst., 16(3):872–923, 1994.

[21] v. Lamsweerde, A. "Formal specification: a roadmap", in Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, 2000, pp.147-159.

[22] Lee, B.-S. and B. Bryant, Automated conversion from requirements documentation to an object-oriented formal specification language, in: SAC'02: Proceedings of the 2002 ACM symposium on Applied computing (2002), pp. 932–936.

[23] Liaskos S., Lapouchnian, A., Yu, Y., Yu, E,. Mylopoulos, J., On Goal- based Variability Acquisition and Analysis, in: 14th IEEE International Requirements Engineering Conference, Minneapolis, USA, 2006

[24] Liaskos S. (2008) Acquiring and Reasoning about Variability in Goal Models, dissertation, University of Toronto.

[25] Mikkonen, T. A layer-based formalization of an on-board instrument. Technical Report 18, Tampere University of Technology, Software Systems Laboratory, 1998.

[26] Niu, N. and Easterbrook, S. Extracting and Modeling Product Line Functional Requirements, in: RE'08, Barcelona, Spain, 2008, pp. 155-164.

[27] Nummenmaa, T., Kuittinen, J., and Holopainen, J. Simulation as a game design tool. In ACE '09: Proceedings of the International Conference on Advances in Computer Entertainment Technology, pages 232–239, New York, NY, USA, 2009. ACM.

[28] Pohl, K., The three dimensions of requirements engineering: a framework and its applications, Information Systems 19 (3) (1994) 243-258

[29] Rolland, C. and Achour, C. B.. Guiding the construction of textual use case specifications. Data & Knowledge Engineering, 25(1-2):125-160, 1998.

[30] van den Brand, M.G.J., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L. Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E. and Visser, J. The ASF+SDF Meta-environment: A component-based language development environment, Proceedings of the 10th International Conference on Compiler Construction, p.365-370, April 02-06, 2001.

[31] Veijalainen, J., Berki, E., Lehmonen, J. & Moisanen, P. Realising a New International Paper Mill Efficiency Standard - Using Computational Correctness Criteria to Model and Verify Timed Events". Eleftherakis, G. (Ed) The 2nd South-East European Workshop on Formal Methods. Practical dimensions: Challenges in the business world. 18-19 Nov 2005, Ohrid. Satellite of the 2nd Balkan Conference in Informatics, Ohrid, FYROM, 17-20 Nov 2005.

[32] B. Whittaker, What went wrong? Unsuccessful information technology projects, Information Management & Computer Security, 7(1), pp. 23-29, 1999P.

[33] Zave and M. Jackson, Where Do Operations Come From? A Multiparadigm Specification Technique, IEEE Transactions on Software Engineering, Vol. 22 No. 7, July 1996, 508-528.