Alloy as a Language for Domain Modeling

Nesa Asoudeh, Ramtin Khosravi
School of Electrical and Computer Engineering,
University of Tehran, Kargar Ave., Tehran, Iran.
{n.asoudeh, rkhosravi}@ece.ut.ac.ir

Abstract- **Domain model is a rigorously organized and selective abstraction of a domain expert's knowledge of his job. Capturing domain models needs a suitable formalism as the modeling language that clearly reflects the concepts in the domain model and is capable of providing feedback to the modeler during domain analysis. Alloy is a light weight modeling language based on first-order relational logic. In this paper we discuss how this language can be used in domain modeling. Using Alloy which supports software abstractions we can start from a simple and small model of the domain and extend it gradually. At each step Alloy analyzer will give us immediate feedback automatically. Instances of our model can be useful in detecting the defects of it and therefore achieving a better understanding of user requirements.**

I. Introduction

One of the main activities of any development process aimed at creating useful software is domain modeling. Domain is the business processes being automated or the real world problem being solved by a software program. Having a good understanding of a software domain is essential for the success of any software development project.

Software domains, being part of the real world, are hard to be captured directly. They have to be modeled using appropriate abstractions. Because of the great amount of knowledge related to any subject area, a domain model has to be a selection of that knowledge and unnecessary details have to be eliminated. Selection and abstraction of this knowledge requires communication between software specialists and the domain experts, who are specialists in the subject area of the software being developed. More about domains and domain modeling process is presented in Section II.

A domain model can be represented in many ways, diagrams, text, etc. but a successful modeling paradigm is the one in which domain experts can be involved as well.

Alloy is a lightweight modeling language, based on first order relational logic [10]. It is supported by an efficient tool called Alloy Analyzer [1]. Section III of this paper is a brief introduction to this modeling language.

In this paper, we are going to present how Alloy can be used as a language for domain modeling. Usually achieving a correct and complete domain model at the initial step is not possible. A good understanding of the domain is result of several iterations. Alloy analyzer, using the SAT technology, can provide us with instances of our model. The instances it finds can help us get a deeper insight into the domain after each iteration.

Alloy does not have a complex mathematical syntax and models provided using this language are easy to understand. Fully automatic analysis performed by its analyzer and visualizations provided by that tool as well as the support for abstractions make this language a good choice for domain modeling.

In Section IV, we are going to explain how this language can be used during the domain modeling process using a case study on university information system. Having applied alloy to domain modeling process, we are going to discuss its strengths and weaknesses as a domain modeling language.

Alloy as a modeling language has been used in a wide range of software engineering research studies. In Section VI we are going to mention some of these studies as related works.

II. Domain modeling

Any software program has a subject area to which its users are going to apply the program. This subject area is referred to as the Domain of the software. Useful software can not be developed without considering its domain. In order to control the inherent complexity of problem domain no one can avoid a deep understanding of the domain itself.

Software domains whether tangible or intangible are related to the real world. Software domains, except for cases such as CASE tools, have little to do with the computers [7]. It is not possible to turn them into code instead, we have to create abstractions of domain. These abstractions are called models. A model is a simplification. It abstracts the aspects related to the problem at hand and ignores unnecessary details.

A domain model is not a particular diagram but the idea that this diagram is going to convey. A Domain model does not have to include all the knowledge that a domain expert has about his job. It should be a selective abstraction of that knowledge [7]. This abstraction may be presented by diagrams as well as an English sentence or even carefully written code.

Considering these facts there is no standard notation or language for domain modeling. The main point is achieving a model that is deeply rooted in user requirements and activities. In order to have such a deep understanding of the requirements, domain experts should be involved in the process of domain modeling. Ideas of domain experts must be communicated during an iterative process, because it is almost impossible to have a complete understanding of the domain initially.

A modeling language that allows starting with a small and simple model and then developing it incrementally using several levels of abstraction can be a good choice for domain modeling.

III. ALLOY

Alloy is a lightweight modeling language based on the first order relational logic [10]. Each Alloy model consists of one or more modules. In an Alloy module, there are a number of signatures which define sets of atoms. The definition of a signature may contain a number of fields which define relations between atoms of signatures. Signatures also serve as types, and subtyping is possible through signatures extension. Signature extensions define new signatures as subsets of the main signature.

There are also ways to define constraints in the model, using constraint paragraphs. There are four kinds of constraint paragraphs:

- *Fact*: A constraint that always holds
- *Predicate*: Named and parameterized formulas that can be used elsewhere
- *Function*: Named and parameterized expressions that can be used elsewhere
- *Assertion*: A constraint that is intended to follow from the facts of a model

Note that facts can be defined in two ways: either following a signature declaration, or elsewhere in the model. In the first case, they are implicitly quantified over all atoms of the signature.

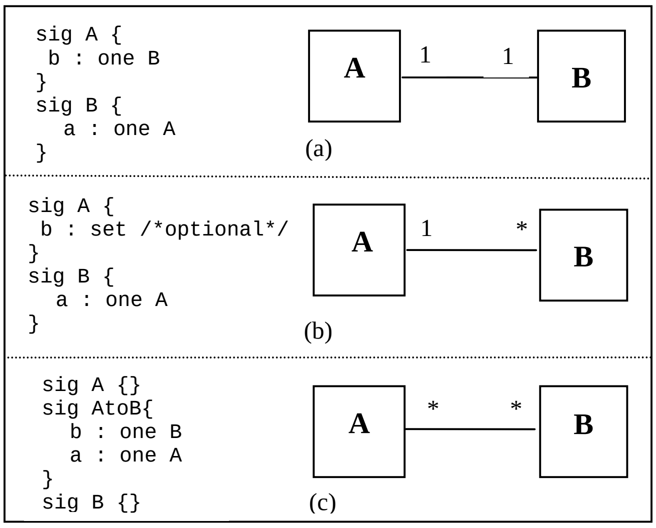
A model in Alloy means a collection of instances. Instances are binding of values to variables. The Alloy Analyzer finds instances of a model automatically by assigning values to variables satisfying the constraints defined. Model analysis involves constraint solving, and the analyzer embodies a SAT solver. It provides visualization for making sense of solutions and counterexamples it finds [11].

Instructions to Alloy Analyzer to perform its analysis are called commands. A run command causes the analyzer to search for an instance that witnesses the consistency of a function or a predicate. A check command causes it to search for a counterexample to show that an assertion does not hold. Searching for instances is done within finite bounds, specified by the user as scope. So, when the search fails, it does not mean that there is no instance satisfying the model (i.e., the model is inconsistent).

IV. DOMAIN MODELING USING ALLOY

In this section, we show how Alloy can be used in domain modeling. We are going to explain the procedure using an example. This example which is a case study performed on a university information system can be useful in making the concepts clear. Also trying to replace common languages for domain modeling by Alloy can help us to reveal its strengths and weaknesses in domain modeling compared to other languages.

This case study was performed in two phases and each phase consists of some iterations.



```
sig A {
 b : one B
}
sig B {
  a : one A
}
```

```
sig A {
 b : set /*optional*/
}
sig B {
  a : one A
}
```

```
sig A {}
sig AtoB{
  b : one B
  a : one A
}
sig B {}
```

Fig. 1. Modeling associations in Alloy

In the first phase, our goal was to model people and educational units in the university as well as their relationships and constraints regarding these relationships. In the second phase, we added educational concepts to the domain model. At the end of each iteration, domain model and its visualized instances were presented to domain experts to check the correctness of the model as well as getting their ideas for the next iteration.

The first domain model was a small one covering a limited portion of the domain concepts. This model was completed gradually using several levels of abstractions.

At first, concepts related to people (professors, students, ...) and educational units (university, campus, department, ...) and their relationships were modeled.

In our approach, entities are modeled using the concept of signature in Alloy. Therefore, each entity will be represented by a set and instances of the entities will be represented by the members of their corresponding set.

We model associations as relations between signatures. Associations with different multiplicities can be modeled this way. Figure 1 shows how one-to-one (a), one-to-many (b) and many-to-many (c) associations are modeled in Alloy.

Inheritance is modeled using the *extends* or subsignature relation between signatures. Following statements demonstrate how to declare B as a child of A.

```
sig A {}                    sig B extends A{}
```

Business rules and domain specific constraints are defined using Alloy facts. Therefore, these constraints are defined as conditions that are always true. For example, to define a constraint saying that in each department vice chairman for student affairs and vice chairman for graduate studies positions should be held by different persons we can write the following Alloy fact:

```
fact business_rules{
  all d: Department |
  (d.studentAffairsVC != d.graduateStudiesVC)}
```

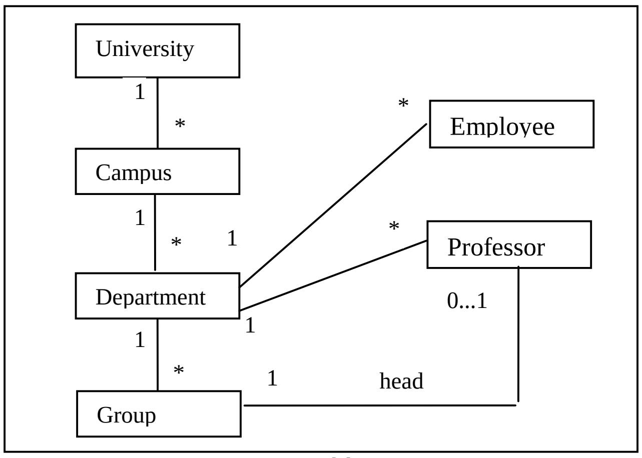


Fig. 2. Domain model using UML notation

As mentioned before, at the first step our goal was to model different places at university as well as people working at these places. This model also included the hierarchical structure of the university; each university consists of some campuses each of them having some department with some different groups defined in each department.

In addition to employees working at their corresponding departments there are certain managing committee positions (chairman and vice chairmen) defined at each department, campus and university which are held by their corresponding faculty members.

The following Alloy module is the domain model covering the facts mentioned above which are presented in Figure 2 using UML notation.

```
module university

sig Professor {
   worksAt : one Department ,
   group : one Group
}
sig Employee { worksAt : one Department}
sig Student {}
sig University {
   chairman : one Professor ,
   educationalVC : one Professor ,
   researchVC : one Professor ,
   studentAffairsVC : one Professor ,
   graduateStudiesVC : one Professor
}
sig Campus {
   belongsTo : one University
   chairman : one Professor ,
   .../* the same as positions in university */
}
sig Department {
   chairman : one Professor ,
   .../* the same as positions in university */
}
sig Group {
   head : one Professor ,
  belongsTo : one Department
}
```

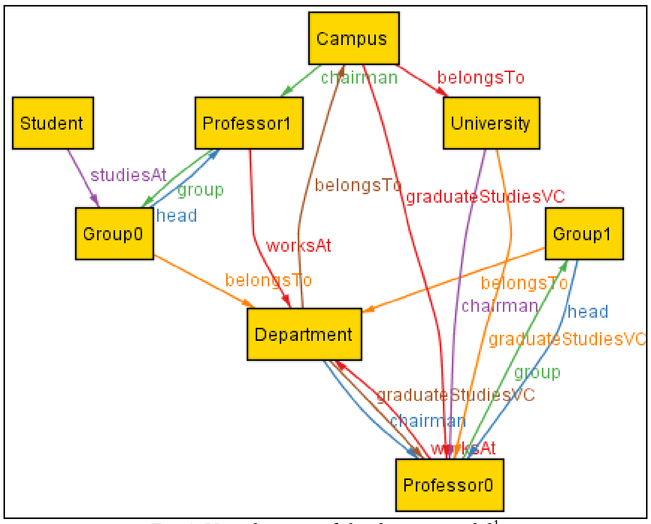


Fig.4. Visualization of the domain model[1]

Consulting the above model and its visualized instance presented in Figure 3 with domain experts (faculty members and employees) revealed the fact that no professor can hold two managing committee positions at the same time. Therefore the following fact about university was added to the model:

```
fact managing committee rules{
  all u:University |
    (u.chairman != u.educationalVC) &&
    (u.chairman != u.researchVC) &&
     (u.chairman != u.studentAffairsVC) && ...}
```

The same facts about Campuses and Departments were added to the model.

As it is clear from the first version of the domain model, managing committee has the same structure in all educational units (university, campus, department) so an abstract signature, called EducationalUnit, was defined which contained the definition of this managing committee and all the units were defined as its children. This abstraction eliminated the similar definition of managing committee in all units.

```
abstract sig EducationalUnit{
   chairman : one Professor ,
   educationalVC : one Professor ,
   ...
}
sig University extends EducationalUnit{}

sig Campus  extends EducationalUnit{
   belongsTo : one University
}
sig Department extends EducationalUnit{
   belongsTo : one Campus
}
```

This abstraction also caused a change in the facts about business rules. Similar statements about University, Campus and Department were replaced by the one for all educational

[1] The figure contains a simplified version of the model. There fore, some parts of it are eliminated.

units. An assertion was written which checked that facts stated before the abstraction were also true after the abstraction.

At the end of the second iteration domain model was presented to domain experts again. Visualizations provided by Alloy analyzer (such as the one demonstrated in Figure 3) helped them in having a clear understanding of the model. They wondered if the model could support changes in the structure of the managing committees also, they believed that in future addition of new positions to the committees is possible. As a response to this new requirement, a new abstraction was added to the model during the third iteration: an abstract signature called Role was added to the model and all positions in managing committee were defined as its children. With this new abstraction, addition of new positions can be performed in a flexible manner, all we will have to do is defining a new child for the Role signature.

```
abstract sig Role {}
one sig EducationalVC extends Role{}
one sig  ResearchVC extends Role{}
...

abstract sig EducationalUnit{
   vcs : Role  -> one Professor
}
```

Having prepared a relatively consistent model of people and educational units, educational concepts such as course offering, registration, curriculums, assessment,... were added to the model. Following Alloy signatures and facts were added to the previous model to cover the concepts related to educational affairs:

```
sig Grade {  value : one Int  }

sig Student {
   passed : set Course ,
   average : one Grade
}
sig Course{ preReqs : set Course}

sig Semmester{}

sig Offering{
   offeredBy : one Professor ,
   offeredAt :one Semmester ,
   offeredIn : one Group ,
   offered : one Course ,
   capacity : one Int ,
   group : one Int ,
   enrollments : set Enrollment
}
sig Enrollment{
   enrolled : one Offering ,
   enrolls : one Student
}
sig Assesment{
   enroll : one Enrollment ,
   grade : one Grade
}
fact educational facts
  all g : Grade |
    lte[g.value,4] && gte [g.value , 0]

   --Min allowed average for enrollment
  all e : Enrollment| gte[e.enrolls.average.value,2]

   --checking prerequisites
   all e : Enrollment , c : Course|
     c in e.enrolled.offered.preReqs implies
     c in e.enrolls.passed

  all o: Offering| #(o.enrollments) < o.capacity

  all o : Offering | o.offeredIn = o.offeredBy.group
}
```

This new version of the model was consulted with domain experts again and the following constraints were revealed:

1. Courses offered by the same professor should not have the same offering time in the same semester.
2. Students should not enroll the same course more than once in a semester.
3. Students should not enroll the courses they have already passed.
4. Each course should be offered by a specified group
5. There must be a default semester for each course and courses having the same default semester should not have the same offering time. So that students have the opportunity to enroll all of them during one semester.

Time and OrdinalSemester signatures as well as following facts were added to the model to cover the constraints mentioned above. They are numbered according to the constraints they represent.

```
1.all disj o,o' : Offering |
  (o.offeredBy == o'.offeredBy implies
 o.time != o'.time)

2.all disj e , e' : Enrollment |
  (e.enrolled.offered ==e'.enrolled.offered)
  implies e.enrolls != e'.enrolls

3.all e : Enrollment |
  !e.enrolled.offered in e.enrolls.passed

4.all o : Offering | o.offeredIn = o.offeredBy.group

5.all disj o,o' : Offering |
      o.offered.defaultSem ==o'.offered.defaultSem
         implies o.time != o'.time
```

During the last iteration concept of curriculum was added to the model. Each curriculum corresponds to a specified group. Prerequisites for courses and their default semester are defined in the curriculum corresponding to the group in which they are offered.

```
sig Curriculum{
   group : one Group,
   preReq   : Course one -> set Course ,
    defaultSem: Course one ->one OrdinalSemester
}
```

Addition of this new concept caused some changes in the facts. Again, consistency between different versions of the model was checked using Alloy assertions.

V. Discussion

In Section 4, we explained how Alloy can be used during the domain modeling process. Using this language and its efficient tool, Alloy analyzer, we could complete the domain model gradually and at each step we could consult the visualized instances of our model with domain experts and correct our model using the feed back we got from them. Presenting the visualized model to domain experts also resulted in the creation of new ideas for next iterations.

Using Alloy as a domain modeling language we did not have to prepare an elaborate model containing all the details initially. This is one of the main benefits of Alloy that its tool does not require an elaborate model to perform its analysis [10]. Also, in order to test a model there is no need to its implementation. Alloy models can be tested at any time during the software development process, even during the early ones such as, requirements engineering phase. All we have to do is providing the correctness properties of our model and then Alloy analyzer will perform its task automatically.

Having a simple syntax, compared to other formal languages, Alloy models are easily understood by the domain experts. Therefore they were effectively involved in the requirements engineering and helped the developer to achieve a clear understanding of the domain.

Alloy analyzer provides us with an immediate result of its analysis, therefore at each step we could correct our model based on the feedback. Counterexamples provided by this tool were very helpful in detecting the faults in our model. These instances and counterexamples were helpful in getting a deep insight to the domain.

Alloy is a modeling language based on relational logic, so pure object oriented concepts, such as inheritance, are not included in this language directly. Therefore, in order to use these concepts in Alloy models we had to find a way to simulate them in this language. Some of these solutions were mentioned at Section IV.

These solutions are just simulations of the Object Oriented concepts, so they can not be in accordance with their corresponding OO concepts in all manners. For example we modeled inheritance with Alloy subsignatures but as it was mentioned in part III atoms of subsignatures fall in a subset of their parents so they can not override the fields of their parents. This fact can cause some limitations in certain cases.

As an example, a new requirement in our case study could be the concept of dual degree programs. Students participating in these programs can study in more than one group so we should override the multiplicity of the group filed of the signature modeling these students. But this is not possible in Alloy and all we can do is adding a new field to the signature representing them.

```
sig Student {
    passed : set Course ,
    average : one Grade ,
    group : one Group
}
sig TwoGroupSt extends Student{
    secondaryGroup : one Group
}
```

Alloy does not provide a strong set of primitive types. There are no character strings or floating-points in this language and we had to define new signatures for them which added to the size and complexity of our models.

VI. Related Works

Alloy as a modeling language has been used in a wide range of software engineering research studies. The aim of most of these case studies was using Alloy and its tool for verifying and analyzing software systems. Following is some categories of the research performed by now:

- Using Alloy in model driven development: this includes using Alloy in verifying MDA transformations [3] and verifying models developed using other languages, such as UML, by transforming them to Alloy models [5],[2].
- Comparing Alloy with other modeling languages such as Z or UML/OCL [13], [16], [9]
- Using Alloy in business process modeling: Current business process modeling tools and languages can only perform a syntactic check on the form of model. Alloy can be used to check the semantics of the business process models [17].
- Alloy as an Architecture Description Language (ADL): Alfa is an extensible composition framework for architectural styles. This framework is developed using Alloy modeling language [10].
- Alloy as a formal verification tool: Alloy as a formal language has been used to verify a great number of protocols and mission critical systems [15], [17].

The works listed above have used Alloy as a formal language for the verification of the models but, in this paper we claim that this language can be used as a model finder as well. We have used this capability to create instances of domain models and present them to domain experts.

As mentioned before, a domain model can be constructed using many approaches. Some of these approaches are using Domain Specific Languages (DSLs) or using well known languages such as UML/OCL.

There are two classes of DSLs. *External* DSLs are written in a different language than the main (host) language of the application and are transformed into it using some form of compiler or interpreter. XML configuration files are instances of this category. Using *Internal* DSLs domain specific concepts are embedded in to the code written in the host language [8].

Domain models prepared using external DSLs are easily understood by domain experts but getting instances of them and assurance of their correctness and consistency is not possible until we implement them using the host language. Deferring testing and verification to the implementation phase can change even small errors in to serious bugs.

Domain models written in internal DSLs, although being executable, are not easily understood by domain experts and they can not be effectively involved in the domain modeling process.

Domain models can also be prepared using well known languages such as UML/OCL. Domain models should be presented using the predefined classes of diagrams (Class Diagrams, Activity Diagrams, etc.) and their corresponding features.

Getting instances of the models constructed using UML/OCL is not possible until they are implemented. Also, because of the predefined functionality of different classes of UML diagrams, structural and behavioral aspects of the model have to be presented using different diagrams (for example, class diagrams and sequence diagrams) and they can not be presented by a single UML diagram. But, Alloy is able to handle both structural and behavioral aspects in the same model [16] and this is useful in having a better understanding of the domain.

VII. CONCLUSION AND FUTURE WORK

In Section 4, we presented how Alloy can be used as a domain modeling language. Alloy, like any other modeling language, has strengths and weaknesses as a domain modeling language, some of which were mentioned in Section 5. There are alternate solutions for some of the Alloy weaknesses. The problem with overriding could be solved by adding a new field to the child's fields, but some of these problems are harder to fix. For example, using the current tool provided for this language, Alloy models are not scalable enough and in order to get instances of our model, so we have to divide it into separate modules.

As mentioned in Section 6, using Domain Specific Languages is another approach to domain modeling. Using Alloy as a domain modeling language, we can have the strengths of both classes of Domain Specific Languages (internal DSLs and External DSLs) at the same time. Alloy models, easily visualized by Alloy Analyzer, are easily understood by the domain experts and they are executable as well, meaning that we can have instances of them and check their correctness using assertions. This approach to verification does not need any test cases and as mentioned in Section 6, models constructed using this approach can present both behavioral and structural aspects of the domain at the same time.

Finally, Alloy models are concise due to its formal nature and if we manage to present the domain at hand using small modules with low level of dependency between them, instances of our model found by Alloy analyzer can be very helpful in having a deep and concise understanding of the model.

Software systems with high level of usability are result of a correct transformation from domain model that is deeply rooted in user requirements into code. Our future goal is transforming Alloy models into code. That means finding equivalences of Alloy models in commonly used programming languages such as Java, ...

REFERENCES

[1] Alloy Analyzer, Available from: http://alloy.mit.edu.

[2] K. Anastasakis, B. Bordbar, G. Georg and I. Ray, "UML2Alloy: A Challenging Model Transformation", ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS), 2007.

[3] K.Anastasakis, B. Bordbar, J. M. Käuster, "Analysis of Model Transformations via Alloy", Modevva, 2007.

[4] F. V. Barajas, "A formal model for a requirements engineering tool", First Alloy workshop, Portland, Oregon, USA, 2006.

[5] B. Bordbar and K. Anastasakis, "UML2Alloy: A tool for lightweight modelling of Discrete Event Systems", IADIS International Conference in Applied Computing, Algarve, Portugal, 2005, pp. 209-216.

[6] B. Bordbar and K. Anastasakis, "MDA and Analysis of Web Applications", Proceeding of VLDB Workshop on Trends in Enterprise Application Architecture (TEAA 2005), Lecture Notes in Computer Science, Vol 3888, pages 44-55.

[7] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003.

[8] M. Fowler, "Language Workbenches: The Killer-App for Domain Specific Languages?", June 2005, available from: http://www.martinfowler.com/articles/languageWorkbench.html .

[9] Y. He, "Comparison of the Languages Alloy and UML", Software Engineering Research and Practice (SERP), 2006.

[10] D. Jackson, Software Abstractions, Logic, Language, and Analysis, MIT Press, 2006.

[11] D. Jackson, "Alloy 3.0 Reference manual", http://alloy.mit.edu, May 10 2004.

[12] D. Jackson, "Alloy: A lightweight object modeling notation", ACM Transactions on Software Engineering and Methodology (TOSEM'02). Volume 11, Issue 2 (April 2002), pp. 256-290.

[13] D. Jackson, "A Comparison of Object Modelling Notations:Alloy, UML and Z", MIT press, 1999.

[14] N. R. Mehta, N. Medvidovic, "Distilling Software Architectural Primitives from Architectural Styles", ESEC-FSE, 2003.

[15] M. Taghdiri and D. Jackson, "A Lightweight Formal Analysis of a Multicast Key Management Scheme", Proc. of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE), October 2003, pp 240-256.

[16] M. Vaziri and D. Jackson, "Some Shortcomings of OCL", TOOLS, 2000.

[17] C. Wallace, "Using Alloy in Process Modelling", Information and Software Technology, 2003.

[18] I. Warren, J. Sun, S. Krishnamohan and T. Weerasinghe, "An Automated Formal Approach to Managing Dynamic Reconfiguration", 21st IEEE/ACM International Conference on Automated Software Engineering (ASE), Tokyo, Japan, September 2006.