Generating AMF Configurations from Software Vendor Constraints and User Requirements

A. Kanso[1], M. Toeroe[2], A. Hamou-Lhadj[1], F. Khendek[1]
[1]*Electrical and Computer Engineering Department*
Concordia University, Montréal, Canada
{al_kan, khendek, abdelw}@ece.concordia.ca
[2]*Ericsson Inc., Montréal, Canada*
Maria.Toeroe@ericsson.com

Abstract

The Service Availability Forum (SAF) has defined a set of service API specifications addressing the growing need of commercial-off-the-shelf high availability solutions. Among these services, the Availability Management Framework (AMF) is the service responsible for managing the high availability of the application services by coordinating redundant application components. To achieve this task, an AMF implementation requires a specific logical view of the organization of the application's services and components known as an AMF configuration. Developing manually such a configuration is a complex, error prone, and time consuming task. In this paper, we present an approach for automatic generation of AMF configurations from a set of requirements given by the configuration designer and the description of the software as provided by the vendor. Our approach alleviates the need of configuration designers dealing with a large number of AMF entities and their relations.

Keywords: High availability, Redundancy models, Service Availability Forum, Availability Management Framework, Configuration generation.

1. Introduction

High availability of a software system is achieved when the system's services are accessible (or available) to its users 99.999% of the time [1]. Availability of a system depends mainly on its reliability and on the extent to which it can be easily repaired. Since ensuring reliability is not always possible, the work towards sustaining highly available systems has shifted towards improving their reparability. The reparability of a system can be significantly improved by adding redundant components that can take over the services provided by faulty components [1].

The Service Availability Forum (SAF) [2] is a consortium of telecommunications and computing companies working together to define and standardize high availability solutions for systems and services. SAF has developed an Application Interface Specification (AIS), which includes the Availability Management Framework (AMF) [3]. The role of AMF is to manage the availability of the service provided by an application. This is achieved through the management of its redundant components and by shifting dynamically the workload assigned to a faulty component to a redundant and healthy one when a fault is detected.

The AMF service requires a configuration for any application it manages. An AMF configuration can be seen as an organization of some logical entities. Designing such an AMF configuration requires a good understanding of AMF entities, their relations, and their grouping. The grouping of entities is guided by the characteristics of the application implementation that will be deployed in an AMF managed cluster. These characteristics are described by the software vendor in terms of types delivered in an Entity Types File (ETF) [4].

Designing an AMF configuration consists of specifying a set of AMF entity types and their entities from a set of ETF types, in order to provide and protect the services as requested by the configuration designer. Creating manually such a configuration can be a tedious and error prone task due to the large number of required types, entities and their attributes. This is combined with the complexity of selecting the appropriate ETF types and deriving from them the necessary AMF types to be used in the configuration.

During the type selection and derivation, several constraints need to be satisfied. Some of these constraints require calculations and extensive consistency checks.

In this paper, we present a technique for the automatic generation of AMF configurations that satisfies all AMF and ETF constraints. We describe an algorithm to select the appropriate ETF types and the method of deriving the appropriate AMF types. The configuration generation completes with populating the configuration with entities of the AMF types.

The remaining part of this paper is organized as follows: In Section 2, we introduce AMF and the concepts needed to understand this paper. In Section 3, we introduce our approach of generating AMF configurations, followed, in Section 4, with a discussion summarizing the main issues and lessons learned. The related work is reviewed in Section 5. We conclude in Section 6 and present some future directions.

2. Availability Management Framework

In clustered systems, applications may be composed of several processes running on different nodes. The role of the AMF service is to protect the services provided by these applications by managing the redundancy of their processes. For this purpose, these application processes need to be organized and described according to the following logical view.

The smallest entity AMF manages is the component, which represents a set of hardware and/or software resources. AMF manages each component through API calls implemented by a dedicated process representing such a component. A service unit (SU) is a logical entity that consists of one or more components that collaborate with each other to combine their services. The workload associated with providing some service, which can be assigned to the component is represented by a component service instance (CSI). A set of CSIs assigned to components of the same service unit is represented by a service instance (SI). Thus, the SI is the workload that is assigned to an SU by AMF at runtime. It also represents the combined higher level service of the collaborating components within the SU.

AMF maintains the availability of the SIs by managing their assignments among a set of redundant SUs. For this purpose, SUs are grouped into service groups. A service group (SG) is a set of SUs that collaborate to protect a set of SIs. An SG protects SIs according to a certain redundancy model. There are five different redundancy models: 2N, N+M, N Way, N Way Active, and No-Redundancy. These redundancy models vary depending on the number of SUs that can be active and standby for the SIs and how these assignments are distributed among the SUs. For example, in the 2N redundancy model one SU is active for all the SIs protected by the SG and one is standby for all the SIs. Whereas, in the N + M model, N SUs share the active assignments and M share the standbys. Both models allow at most one active and one standby assignment for each particular SI. An AMF application is composed of one or more SGs.

There are two additional AMF logical entities used for deployment purpose: The cluster and the node. The cluster consists of a collection of nodes under the control of AMF.

Figure 1 illustrates an example configuration of an application and the services it provides. The application consists of one SG with 2N redundancy model protecting two SIs. Here, SU1 is active for both of them and SU2 is standby.

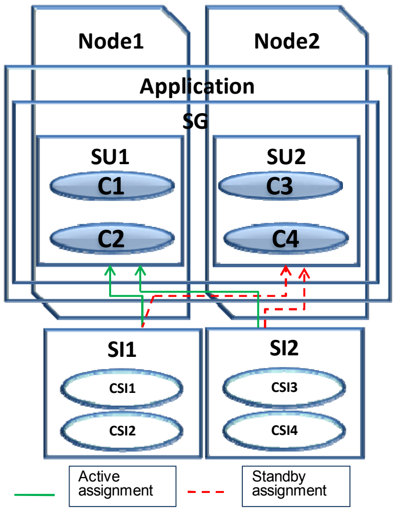

Figure 1. An example of AMF configuration and its services

All AMF entities, except the cluster and nodes, are typed. The entity types provide AMF with the information about the shared characteristics of their entities. For example, the component type determines the component service types any component of this type shall provide. AMF uses this information to determine to which component within an SU to assign a particular CSI. Therefore, when building an AMF configuration, we need to determine the AMF types before creating the AMF entities of these types.

AMF types are derived from types defined in the Entity Types File (ETF) [4], which is an XML file provided by the software vendor and that describes the application software developed to be managed by an AMF implementation. ETF types, their relations, and main characteristics can be summarized as follows.

Component Type: A component type describes a particular version of a software implementation designed to be managed by AMF. The component type specifies the component service types that the components of this type can provide. It defines for each component service type the component capability model and any required dependency on other component service types. The component capability model is defined as triplet `(x, y, b)`, where `x` represents the maximum number of active CSI assignments and `y` the maximum number of standby CSI assignments a component can handle for a particular component service type. While `b` indicates whether active and standby assignments can be handled simultaneously.

Component Service Type (CS type): It describes the set of attributes that characterizes the workload that can be assigned to a component of a particular type in conjunction of providing some service.

Service Unit Type (SU type): The service unit type specifies the service types an SU of the type can provide, and the set of component types from which an SU of the type can be built. It may limit the maximum number of components of a particular type that can be included. Thus, the SU type defines any limitation on the collaboration and coexistence of component types within its SUs.

Service Type: A service type defines the set of component service types from which its SIs can be built. The service type may limit the number of CSIs of a particular CS type that can exist in a service instance of the service type. It is also used to describe the type of services supported by an SU type.

Service Group Type (SG type): The service group type defines for its SGs the redundancy model. It also specifies the different SU types permitted for its SGs. Thus the SG type plays a key role in determining the availability of services.

Application Type: The application type defines the set of SG types that may be used to build applications of this type.

It should be noted that the only mandatory types in an ETF accompanying a software implementation are the component type and the component service type. Further types (i.e., SU types, SG types, service types and application types) may be used by the vendor to specify software implementation constraints and limitations to be considered at the integration of this software with an AMF implementation, but they are not mandatory. If no optional type is provided at a particular level (e.g. there is no SU type), then types of the level below (i.e. the component types) can be combined as needed without any restriction. If any optional type is provided at a particular level, it is assumed that all allowed types at that level are specified by the vendor.

3. AMF Configuration Generation Method

The configuration generation method takes input from two sources: The configuration designer and the software vendor.

The configuration designer defines the services to be provided in terms of SIs and CSIs together with their desired protection (i.e., the redundancy model); as well as the cluster of deployment and its nodes (e.g., the number of nodes). Based on the latter information, the AMF cluster entity and the node entities can be added to the configuration right away.

To facilitate the definition of services, templates were defined: An SI-template represents the SIs that share common features including the service type, the desired redundancy model: (e.g., 2N, N+M, etc.), the number of active and standby assignments, and the associated CSI-templates. A CSI-template defines the CS type with its number of CSIs.

The software to be used to provide these services is characterized by each vendor in an ETF as described in the previous section.

Once the services and the software are specified, the configuration generation method proceeds with selecting the appropriate ETF types for each required service. In the next sections, we present how component, SU, SG, and application types are selected.

3.1. Component Type Selection Criteria

To select an ETF component type, we match the CS type of the CSIs as specified in the CSI-template to one of the CS types of the component type. We also need to verify that the component capability model of the type is compliant with the one required by the redundancy model specified for the SI-template the CSI-template belongs to. Depending on the required redundancy model and the component capability model, certain component types may be more suitable than others, while other component types may not be usable at all. For example, a component type that can have only active or only standby assignments at a time can be used in an SG of the 2N redundancy model, but it is not valid for an SG with N-way redundancy model where the components may have active and standby assignments simultaneously.

In addition, if the component type is checked as part of an SU type, the constraint imposed by the SU type on the component type needs to be verified: The maximum number of components in an SU of this SU type must not be exceeded. This limits the capacity of the SU for a component service type. Therefore, we need to ensure that the maximum number of components of the selected component type can provide the capacity necessary for the load of CSIs imposed by the load of SIs that are expected to be assigned to the parent SU.

3.2. Service Unit Load Calculations

The load of SIs each SU needs to handle depends on the total number of SIs the SG has to protect, the protection it needs to provide for each SI (i.e., the number of active and standby assignments), and the number of SUs among which these assignments are distributed.

More precisely, for each SI-template, we determine the minimum expected load of an SU for the active and standby assignments as defined by the redundancy model using Equations 1 and 2, respectively, where:

- `numSUs` refers to the total number of SUs of the SG that will protect the SIs of the template,
- `numActSUs` refers to the number of SUs that have active assignments only,
- `numStdbSUs` specifies the number of SUs that have standby assignments only,
- `numAct` refers to the number of active assignments for each SI,
- `numStdb` refers to the number of standby assignments for each SI, and
- `redMod` is used for the redundancy model specified by the SI template.

$$ActiveLoad = \begin{cases} \mathrm{ceil}\left(\frac{numSIs}{numSUs}\right) & if\ \ redMod = "N\ Way" \\ \mathrm{ceil}\left(\frac{numSIs \times numAct}{numActSUs}\right) & if\ \ redMod = "N\ Way\ \ Active" \\ 1 & if\ \ redMod = "NoRedundan\ \ cy" \\ numSIs & if\ \ redMod = "2N" \\ \mathrm{ceil}\left(\frac{numSIs}{numActSUs}\right) & if\ \ redMod = "N + M" \end{cases}$$

Equation 1. Determining the load of active SI assignments for an SU

$$StandbyLoa\ \ d = \begin{cases} \mathrm{ceil}\left(\frac{numSIs \times numStdb}{numSUs}\right) & if\ \ redMod = "N\ Way" \\ 0 & if\ \ redMod = "N\ Way\ \ Active" \\ 0 & if\ \ redMod = "No\ \ Redundancy" \\ numSIs & if\ \ redMod = "2\ N" \\ \mathrm{ceil}\left(\frac{numSIs}{numStdbSUs}\right) & if\ \ redMod = "N + M" \end{cases}$$

Equation 2. Determining the load of standby SI assignments for an SU

The above equations use the characteristics of the various redundancy models to determine the load in terms of the number of SI assignments an SU is expected to handle. For example, for the N-way redundancy model, the load of active SI assignments for an SU calculated as the total number of SIs of an SG of the SI-template divided by the number of SUs since an SI is assigned active to at most one SU and all SUs may handle active assignments. To guarantee that there is always an SU, which can take over the active load of a failed SU; we may use `numSUs - 1` as the number of SUs. For the same redundancy model, the number of SIs an SU can handle in a standby state (in Equation 2) is the number of standby assignments of the SIs divided by the number of SUs. The number of standby assignments of all SIs is calculated as `numSIs x numStdb` since, in an N-way redundancy model, an SI can have multiple standby assignments.

We apply the same principle to other redundancy models and compute the load of SIs assigned to an SU in both active and standby states.

Once the load of SIs for an SU is determined we proceed with finding the applicable SU types. That is, we match the calculated loads at the CSI level with the capacity provided by the allowed number of components of the component type of the SU types using the component type selection criteria (Section 3.1).

3.3. Selecting SU, SG, and Application Types

To determine the suitable SU, SG, and application types, we propose an algorithm searching from the highest defined level of types (i.e., the application type).

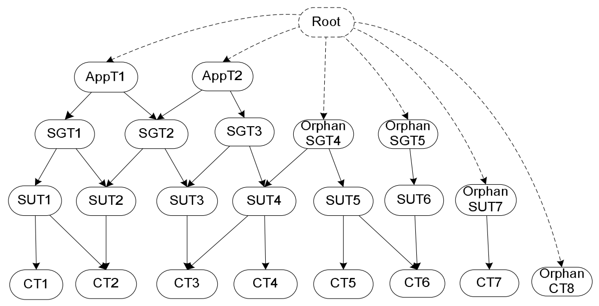

Figure 2. An Example of ETF types.

ETF types can be structured in the form of a disconnected non-ordered directed acyclic graph (DAG) as shown in Figure 2. The nodes of the DAG represent ETF types, whereas the edges represent the relations among these types.

Except for the application type, all other types can have more than one parent type; and since not all types are mandatory in ETF, some types might come with no parents. For example, SG types SGT4 and SGT5 of Figure 2 do not have a parent application type. We refer to these types as *orphan* types.

We added a new node called `Root` to connect the sub-graphs of the DAG to facilitate the type selection algorithm.

Our algorithm for selecting ETF types is given in Figure 3. The function `SelectETFTypes` takes the DAG as input (i.e., its root node) and an SI-template, and returns as output a set, called `Types`, which contains the ETF types - if there are any to provide the services defined by the SI template and that satisfy the constraints that applies to these types, that we refer to as *requirements*. Note that these requirements change depending on the visited type. Moreover, the satisfaction of these requirements may depend on the parent type (e.g. the maximum number of components defined by the SU type and not by the component type), therefore we may need to visit the same node several times.

The initialization step sets `Types` to an empty set. The `SelectETFTypes` function is a recursive function that performs a depth-first search traversal of the DAG starting from the `Root` node to the component types. More precisely, it starts by visiting the first child of the `Root` node (line 1), which in our case, consists of an application type (if provided). It checks if this application type satisfies the configuration requirements (line 2). If this is the case, then it checks if it is a leaf node (line 3) and since it is not it continues by recursively visiting the application type's SG types (line 16). Otherwise, the algorithm proceeds to the next child of the `Root` node which could be another application type, an orphan SG type, an orphan SU type or an orphan component type. If it finds an SG type with a redundancy model identical to the one specified in the SI-template, its SU types are visited.

```
Types = {}
SelectETFTypes(Node N, SI-template SItemp)
1.  For each child of N do
2.    If child satisfies requirements then
3.      If child is leaf then
4.        Types = {child}
5.        ST = {all provided CSTs of child}
6.        If SItemp.ST ⊆ ST then    Return
7.        For each sibling of child do
8.          If sibling satisfies requirements then
9.            Types = Types ∪ {sibling}
10.           ST=ST ∪ {all provided CSTs of sibling}
11.           If SItemp.ST ⊆ ST then Return
12.       Enddo
13.       Types = {}
14.       Return
15.     Else
16.       SelectETFTypes(child, SItemp)
17.       If Types ≠ {} then
18.         Types = Types ∪ {child}
19.         Return
20. Enddo
```

Figure 3. Algorithm for ETF type selection

An SU type is selected if it provides the service type specified in the SI-template and has the capacity (through the components of its component types) to handle the load of SIs (and their CSIs) expected to be assigned to an SU of this type (see Section 3.2). To determine this, the component types of the SU type also need to be visited. When the first component type is reached and satisfies the requirements as described in Section 3.1, it also satisfies the leaf condition (line 3). This component type is added to the selected Types set (line 4), and to verify that service type of the SI template can be satisfied a new set is initialized with the CS types of this child (line 5) and it is checked (line 6) if the required service can be provided. If it cannot be provided yet, each sibling component type is checked (line 7) and if satisfies the requirements, then the type itself and its provided CS types are added to the appropriate sets (lines 8-10). After each sibling we verify if the service type of the SI template can be provided. If so, we return from the component type level (line 11). If we could not satisfy the service type after visiting all the siblings, then we must clear the collected types (line 13) and backtrack to the SU type level (line 14) (or `Root` for orphan components) and select a new SU type.

On the return path depending on whether the Types set is empty or not we either continue the search with the next child (line 20), or add the current child to the Types set (line 18) and return to the level above (line 19). When the algorithm terminates the Types set may contain the following possible sets: (1) a complete type set, i.e. {AppType, SGType, SUType and CompTypes}, (2) some types are missing, e.g. {SUType and CompTypes}, or (3) empty.

In Case 3, the algorithm could not find any types to satisfy the configuration requirements therefore no configuration can be generated to provide the requested services using the software described by the ETF types.

In Case 1, we have all the necessary types. For Case 2, the creation of some missing types is required.

```
CreateTypes(TypeSet Types, SI-template SItemp)
1. lastType = orphan types in Types
2. While AppType ∉ Types do
3.    Build newType from lastType for SItemp
4.    Types = Types ∪ {newType}
5.    lastType = newType
6. Enddo
```

Figure 4. Algorithm for type creation

We use the simple algorithm presented in Figure 4 to create the missing types. It selects the orphan type(s) in `Types` (line 1) and as long as there is no application type in `Types` (line 2) it builds a new type from the last selected type(s) (line 5) according to the SI-template (line 3). In [6], we presented a method that targeted this type creation. Here we restrict that method by allowing type creation only from orphan types.

3.4. Completing the Configuration

Once ETF types are selected (or created), we need to derive the corresponding AMF types, and generate the corresponding AMF entities.

AMF types are created from the selected ETF types by deriving their attributes from the attributes provided in the ETF types. In most cases this means using the default value provided for the type. In other cases, the AMF type is tailored to the intended use. For example, if an ETF component type can provide CST1 and CST2 CS types, but the type was selected to provide only one of them (e.g. CST1), in the AMF type only this component type is listed as provided CS type.

For created types, the AMF type attributes are derived based on the dependencies that can be established among the used ETF types. For example, the default instantiation level of ETF component types of a created SU type is calculated based on the dependency of their provided CS types if any.

Once all the AMF types have been supplied, their entities are created. For this again for each SI-template the number of components is determined within each SU of the selected SU type. This is based on the load of SIs expected to be assigned to the SU (Equations 1 and 2), the number of CSIs within these SIs, as well as the capability of the components, specified in the component type of these components. For example, if the SU can handle one SI, which contains four CSIs in a 2N redundancy model and that the component capability is 2 active and 3 standby then the number of components for this SU should be 4/2 = 2. In other words, we only need two components to be active for the four CSIs. We also need two components of this type in the standby SU to handle the four CSIs (using the ceiling(3/2)).

From the first created SU we duplicate all the necessary SUs to create the SG, which might also be duplicated to satisfy all the SIs of the template. We repeat the process for each SI-template. SGs are combined into applications based on the selected application types.

Next, we determine the distribution of SUs on nodes. According to [3] we assume identical nodes, and as such we distribute SUs in a way that (1) the load of SUs is equally distributed among nodes, and that (2) no two SUs of the same SG reside on the same node (if the number of nodes is sufficient).

The last step is to populate any remaining attributes of the AMF entities. Some of their values come from ETF, whereas others need to be computed in similar manner.

4. Discussion

The issues uncovered during the design of the presented configuration generation method are categorized into three points: Selection of orphan types, selection of higher-level types, and generation of multiple configurations.

4.1. Selection of Orphan Types

The selection of an orphan type requires the population of the attributes of the created parent type. These attributes may require a good understanding of the software implementation and therefore may not be derived automatically. An example of such a type attribute is the component restart probation period found in an SG type. This attribute applies to components in service units belonging to a service group of this type. If the SG type is created (instead of being selected from ETF) then this attribute needs to be determined without any guidance from the vendor. We need to introduce artificially a value, which will affect the software behavior at runtime and therefore impacts the quality of the generated configuration. It is still unclear how to determine safely the attributes of the different entity types when they are created.

A related observation is that it is necessary to maintain the information whether a type was created by our method or provided by the software vendor in the ETF. Created types do not reflect implementation limitation and therefore should not restrict the use of orphan types. However they may be reused whenever they are appropriate to ease the type creation task.

4.2. Selection of Higher Level Types

As a result the preference is to only analyze orphan types when the non-orphan ones have been checked and found unsatisfactory with respect to the configuration requirements. However, there might be situations where we have an ETF (or multiple ETFs) where more than one type can be a candidate. This is typically the case for different versions of software or that offers similar services but provided by different vendors. Some vendors (versions) may constrain the way their components should be configured by having the corresponding component types refer to SU types, while other vendors may provide components that do not need to be constrained from the ETF perspective. In other words, the component types corresponding to these components are orphans. Depending on the required services and the deployment system, there might be situations where orphan component types are a better choice than non-orphan types due to their flexibility. This comes at the price of the difficulties of type creation as aforementioned.

Our technique is easily adaptable to check types in any desirable order.

4.3. Generating Multiple Configurations

Several configurations might be generated from the same set of ETF types and configuration designer requirements. Different sets of types may satisfy the requirements, and also from the same set of types different sets of entities can be created. Finally, the same set of entities may be distributed in the cluster in different manners leading to different configurations.

The algorithm presented in Section 3.3 stops when it finds the first set of types that satisfies the configuration requirements. However, there may be many other possible sets that are candidate solutions. To extract all possible paths that contain valid ETF types with respect to the configuration requirements, the algorithm can easily be adapted by changing the exit condition and saving each path that contains a set of qualified types.

For a single type set, for instance, determining the number of components in an SU is a criterion that can lead to multiple configurations. In the example presented in Section 3.4, we concluded that we needed 2 components in each SU to handle the CSIs associated to these SUs in both active and standby states targeting maximum utilization of the components based on the capability model. However, we could also have created four components of this component type in each SU. This would also have been a valid configuration.

Although there are many configurations that can be generated from the same input sets, not all configurations are equally suitable. There is a need to investigate the criteria by which configurations can be compared. Examples of these criteria include the availability level that a configuration offers, performance, the cost associated with the components, etc.

5. Related Work

The work presented in this paper is part of a larger project, called the MAGIC project [5], which aims at investigating techniques and building tools for automatic generation of AMF configurations for clustered systems as well as the generation of upgrade campaigns for migrating these systems from one configuration to another.

In previous work [6], we presented a first approach for generating automatically an AMF configuration. This approach suffered from some drawbacks: Depending on user preferences, it did not guarantee the generation of valid configurations with respect to ETF types as the creation of types depended on the user preference rather than the "orphan" feature of the selected types. For instance, two component types could be grouped and used in a way not allowed by ETF. The contributions of this paper can be considered as a significant improvement of the work presented in [6]. The contributions consist of a technically sound approach for generating AMF configurations that are valid with respect to a set of ETF types and user requirements.

We are not aware of any other work that focuses on generating AMF configurations. This might be due to the fact that SAF specifications are relatively new and that the implementation of SAF compliant middleware is still an on going effort. Existing middleware implementations such as OpenAIS [7], OpenSAF [8], and OpenClovis [9] do not provide support for the generation of AMF configurations.

Perhaps, the closest research work to our research project is the work presented by Kövi et al. [10], where the authors applied the Model Driven Approach (MDA) to the design of AIS configurations, which were modeled as a Platform Independent Model that they later mapped onto a Platform Specific Model to be used in an implementation of AIS. This study focuses on using MDA concepts to develop applications that can run on an AIS implementation rather than generating AMF configurations.

6. Conclusion and Future Work

In this paper, we present a method for the automatic generation of AMF configurations. Our method ensures a valid configuration with respect to the user requirements and the vendor specifications that come in the form of ETFs. Our method proceeds in a top-down fashion by visiting ETF types starting from the highest ones. We showed the detailed steps involved in our approach including estimating the load of SIs per SU, the selection of component types as well as SU, SG, and application types. We also discussed how AMF types and entities are created once the types are selected or created.

Future work will focus on defining the criteria by which we can compare multiple configurations.

7. Acknowledgements

This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Ericsson Software Research.

8. References

[1] C. Oggerino. *High availability network fundamentals: a practical guide to predicting network availability*. Cisco Press, 2001.
[2] Service Availability Forum™, URL: http://www.saforum.org
[3] Service Availability Forum, Application Interface Specification. Availability Management Framework SAI-AIS-AMF-B.03.01.
[4] Service Availability Forum, Application Interface Specification. Software Management Framework SAI-AIS-SMF-A.01.01.
[5] MAGIC Project, URL: http://users.encs.concordia.ca/~magic/
[6] A. Kanso, M. Toeroe, F. Khendek, A. Hamou-Lhadj, "Automatic Generation of AMF Compliant Configurations", *In Proc. of the International Service Availability Symposium (ISAS), LNCS Vol.5017*, pp. 155-170, Tokyo, Japan, 2008.
[7] OPENAIS, URL: http://www.openais.org
[8] OPENSAF, URL: http://www.opensaf.org/
[9] OPENCLOVIS, URL: www.openclovis.org/
[10] A. Kövi, D. Varró, "An Eclipse-Based Framework for AIS Service Configurations", *In Proc. of the International Service Availability Symposium (ISAS), LNCS Vol.4526*, pp. 110-126, Durham, NH, 2007.