Formal Logic Based Configuration Modeling and Verification for Dynamic Component Systems

Zoltan Theisz
evopro Informatics and Automation Ltd.
Email: zoltan.theisz@evopro.hu

Gabor Batori
Software Engineering Group
Ericsson Hungary
Email: gabor.batori@ericsson.com

Domonkos Asztalos
Software Engineering Group,
Ericsson Hungary
Email: domonkos.asztalos@ericsson.com

***Abstract*—Reconfigurable networked systems have often been developed via dynamically deployed software components that are executing on top of interconnected heterogenous hardware nodes. The challenges resulting from the complexity of those systems have been traditionally mitigated by creative ad-hoc solutions supported by domain specific modeling frameworks and methodologies. Targeting that deficiency, our paper shows that by involving a first-order logic based structural modeling language, Alloy, in the analysis of component deployment we could extend the limits of the generic domain specific metamodeling methodology developed for Reconfigurable Ubiquitous Networked Embedded Systems.**

***Keywords*-Alloy specification; formal model semantics; metamodeling; dynamic component system**

I. INTRODUCTION

Reconfigurable networked component systems provide a versatile platform for implementing highly distributed autonomic peer-to-peer applications in domains of both real sensor networks and autonomic computing [1] environments such as e.g. intelligent network management that relies on sensory and effectory facilities of multi-level control loops. The building and verification of those applications in practice has turned out to be a rather challenging research topic that could enormously benefit from the usage of domain specific modeling approaches. One of the major results of the Reconfigurable Ubiquitous Networked Embedded Systems (RUNES) IST project was to establish a reflective distributed component-based multi-platform middleware architecture [2] for heterogeneous networks of computational nodes, including metamodel-based software development methodology [3] and graphical development framework. The RUNES metamodel provides all the relevant concepts software architects need to efficiently utilize the computational resources within a reflective distributed component-based environment. Due to the inherent complexity of distributed reconfigurable component systems, we advocate the usage of Alloy [4], a formal first order logic based language supported by a fully automated analyzer that has been successfully used to model various complex systems in a wide range of application domains for domain specific model verification purposes. Alloy has been applied in [5] for the analysis of some critical correctness properties that should be satisfied by any secure multicast protocol. The idea of using Alloy for component based system analysis was suggested by Warren et al. [6]. This paper shows OpenRec, a framework which comprises a reflective component model and the Alloy model of OpenRec. This Alloy model served as a basis for our Alloy component model but our model is more detailed which enables deeper analysis of the system behavior. Moreover, [7] demonstrates an Alloy model that identifies the various types of dynamic system reconfigurations. It provides a good categorization of various problems and solutions related to dynamic software evolution. Furthermore, Aydal et al. [8] found Alloy Analyzer one of the best analysis tool for state-based modeling languages.

Although individual application scenarios can be easily expressed manually in Alloy we firmly believe that the synergy between metamodel driven design and first order logic based practical model verification could result in a more advantageous unified approach. Our approach, in a nutshell, semi-automatically generates all the relevant RUNES deployment configuration assets that have also been analyzed within Alloy. By analyzing a significant subset of frequently reoccurring configurations the boundary between valid and invalid component configurations can be thoroughly investigated against proper sets of model-based application and/or middleware feasibility constraints. The analysis results can be used to provide input to the runtime adaptive control logic in order to extend the model-based software development framework [3] with effective autonomicity.

In this paper, we will describe how a first-order logic based model of the RUNES middleware has been developed in Alloy and how it has been integrated into the RUNES domain specific modeling framework and methodology [3]. In the remainder of the paper, first in Section II, we briefly overview Alloy, then, we also disseminate in detail how the RUNES Metamodel has been formalized in it. Next, Section III explains how the Alloy backed verification step gets integrated into the general metamodel-based RUNES application development methodology. Then, Section IV presents a short introduction into the usage how verification results can be incorporated into a full scale model-based

 ISBN: 978-1-61208-130-4

management approach. Next, in Section V, a simplified real-life sensor application will be showcased in order to visualize our approach via a tangible example. Finally, in Section VI we conclude the paper and briefly highlight our future research plans.

II. RUNES METAMODEL VERIFICATION WITH ALLOY

A. Alloy

Alloy [4] is a textual metamodeling language that is based on structured first-order relational logic. A particular model in Alloy contains a number of signature definitions with fields, facts, functions and predicates. Alloy is supported by a fully automated constraint solver, called Alloy Analyzer [9], which can be used to verify model parameters by searching for either valid or invalid instances of the model.

B. Applying Alloy for component system verification

The RUNES reflective component middleware has been created as one of the most important software assets resulting from the RUNES IST project. In general, it consists of a component-based middleware that follows the currently popular loosely-coupled paradigm of Service Oriented Architecture [10]. The middleware is fully reflective, its API is rigorously specified in various programming languages [11], its elements are conceptually backed by multi-layer metamodels and finally a metamodel driven, domain specific application development methodology provides the guidelines for its most effective application. All in all, the RUNES application development is highly streamlined and it is carried out mostly following strict model-based design principles within a semi-automatic metamodeling environment. Although the development techniques and the guiding process have been streamlined, the verification and the validation of the resulting modeling assets such as the application models and the incorporated executable action semantics have to be carried out manually or semi-automatically [12]. Model based test generation proved to be very effective in some scenarios, though testing cannot replace, even in industrial setups, the verification and/or validation efforts. Our industry experience has also proved quite frequently that modeling tasks are highly creative, thought intensive activities which result in complex artifacts of great variability. Therefore, the verification and validation of model-based solutions is a considerable challenge. Fortunately, in reconfigurable sensor applications, the complexity of the resulting system is slightly limited because the variability of the system mainly originates only from the flexibility of the underlying component model of the middleware, thus a more formal way of verifying applications is within practical reach. From the wide spectrum of verification paradigms first-order logic is considered as one of the most rigorous approaches. It has turned out that in our scenarios mostly the independently acting sub-configurations of tightly coupled components have usually caused the majority of the most

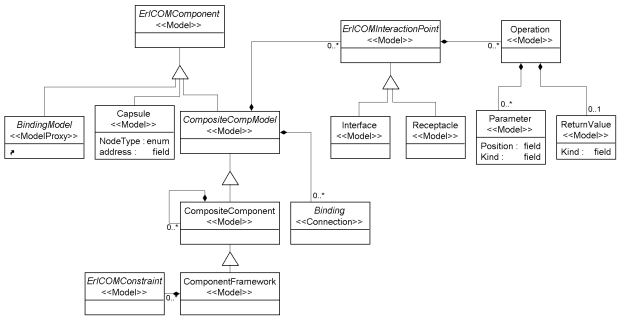

Figure 1. Kernel part of the metamodel

serious malfunctions; therefore, our aim had been mainly directed towards their automatic elimination and avoidance in the runtime deployment. The generic design principles of Alloy [4] facilitates both easy meta-language creation that complies with metamodel driven domain specific language building concepts and formal verification of models. In the following, we will formalize the semantics - from the verification point of view - of our middleware metamodel in Alloy in full accordance with the principle of the semantic anchoring approach reported in [13].

C. Functional metamodel

The RUNES Metamodel specifies the formal metamodel that represents the relevant elements of the RUNES middleware architecture [2]. Figure 1 illustrates the kernel part of the metamodel, which defines the basic concepts of Interfaces, Receptacles, Components and Bindings including their relations and cardinalities. Combined with the associated OCL expressions the RUNES Metamodel establishes a model-based application development environment in GME [14], which enables rapid RUNES application development. In order to be able to verify the proper configuration sequence of a particular modeled application scenario the RUNES Metamodel has to be semantically anchored to a precise structural and behavioral formalism in Alloy. Therefore, the following paragraphs will show how the various metamodeling concepts have been reformulated in Alloy so that application models could be verified against configuration constraints.

In general, the functional specification of any RUNES application must be organized around Components and Bindings. The Components represent the encapsulated units of functionality and deployment. The interactions amongst them take place exclusively via explicitly defined Interfaces and Receptacles. The dynamic behavior of the components are automatically generated from Message Sequence Charts (MSC) and the results are formalized via concurrent Finite State Machines (FSM) [15]. Therefore, a generic RUNES Component is defined as a signature whose fields consist of at most one state machine and a set of Interfaces and Receptacles, respectively.

 ISBN: 978-1-61208-130-4

```
abstract sig Comp{
   state_machine:set StateMachine,
   provided: set Interface,
   required: set Receptacle,
}{
   lone state_machine
}
```

Both the Interface and the Receptacle inherit the common characteristics of an Interaction Point, which is defined by a set of related operation signatures and associated data types. The Interface represents the "provided", the Receptacle the "required" end of a component-to-component connection, respectively.

```
abstract sig Signature{}
abstract sig InteractionPoint {
   signatures: set Signature
}
sig Interface extends InteractionPoint{}
sig Receptacle extends InteractionPoint{}
```

Bindings ensure that connections between Interfaces and Receptacles are set up consistently, according to their proper definitions. Hence, a Binding is defined as a signature that contains fields for one Interface and one Receptacle and one non-identical, component correct mapping that connects the previously mentioned two items.

```
abstract sig Binding{
   mapping:Comp -> Comp,
   interface: one Interface,
   receptacle: one Receptacle
}{
   one mapping
   no (mapping & iden)
   receptacle in (Comp.~(mapping)).required
   interface in (Comp.mapping).provided
}
```

The Receptacle must always represent a requirement which is a 'subset' of the operations (signatures) provided by the Interface it intends to be bound to via the Binding. That fact has to be made explicit in Alloy to allow only correct Bindings in the model.

```
all b:Binding| b.receptacle.signatures in b.interface.signatures
```

D. Deployment metamodel

Figure 2 shows those relevant deployment concepts of the RUNES Metamodel that determine the runtime aspects of a RUNES component application. The key element of the metamodel is the Capsule that represents the generic middleware container, which on the one hand provides direct access to all the functionalities of the runtime API [11], on the other hand it manages a robust fault recovery and redundancy facility. Deploying a component into a capsule in generic terms means that it must be ensured that adequate resources are available for loading in the component in a particular instance of time. The deployed components and bindings might change in time, hence their temporal representation must take into account the explicit definition of Time, too. The Capsule also possesses a distributed, peer-to-peer, fully reflective meta-data repository that can be used for both application and middleware specific purposes. A Capsule in Alloy is defined as a signature having fields representing the temporal evolution of deployed components, bindings and a middleware related resource pool plus the time invariant capsule topology information.

```
open util/ordering[Time] as TO
sig Time{}
abstract sig Capsule {
```

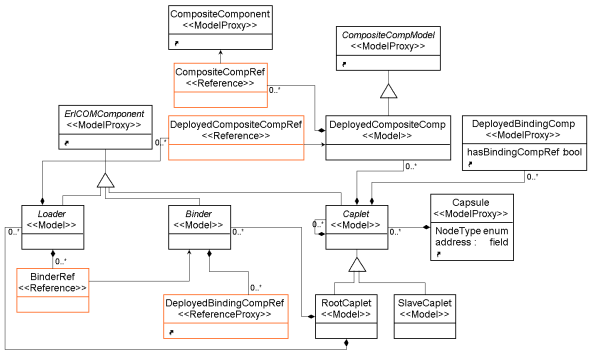

Figure 2. Deployment part of the RUNES Metamodel

```
   comps: DeployedComp -> Time,
   bindings: DeployedBinding -> Time,
   comp_capacity: Int -> Time,
   neighbours: some Capsule
}{
   all t:Time|int[comp_capacity.t] >= #(comps.t)
   all t:Time|comp_capacity.t >= Int[0]
}
```

A deployed component incorporates all the necessary information that stores the active process aspect of the component's functionality including explicit definition of state transitions in time. In other words, the deployed component can be considered as a dynamic instance of a component in accordance with its "ModelProxy" declaration in GME depicted in Figure 2. The temporal aspect of the state transitions are defined by the fire and the current_state fields of the DeployedComp signature.

```
sig DeployedComp{
   deploy: one Comp,
   fire: Transition -> Time,
   current_state: State -> Time
}{
   deploy in FunctionalConf.comps
   all t:Time|lone fire.t
   all t:Time|lone current_state.t
}
```

A deployed binding does not declare time explicitly, however, it contains a mapping field between the two participating deployed components, hence, it is also time dependent. The compatibility of the deployed binding is checked based on the functional definition of the connection.

```
sig DeployedBinding{
   mapping: DeployedComp -> DeployedComp,
   deploy: one Binding
}{
   one mapping
   (DeployedComp.~mapping).deploy =
     Comp.~(deploy.mapping)
   (DeployedComp.mapping).deploy = Comp.(deploy.mapping)
}
```

Our main goal of applying Alloy has been oriented towards configuration verification, thus we must represent a deployed RUNES application in Alloy as a collection of capsules which register the temporal evolution of each of the components and the bindings. Alloy's trace statements help us verify the time evolution of the application against feasibility constraints and the successful runs can also be visualized for human inspection, too.

```
sig DeploymentConf{
   capsules: some Capsule
}
```

The RUNES middleware API supports a set of component management operations such as [un]loading, [un]binding

 ISBN: 978-1-61208-130-4

and migrating components. The operations require time to execute their functionality, they usually modify only local states of the distributed application and keep the rest unchanged. In Alloy, we serialize the potentially concurrent atomic API operations in such a way that one and only one of them can be carried out in one particular instance of time. Due to size constraints only the definition of the migrate operation is presented here in detail. The other operations have been defined applying similar specification techniques.

Component migration is carried out between two capsules by moving an already deployed component between two consecutive points of time. First the preconditions are checked if it is a real migration between two different capsules and there are enough resources available in the receiving capsule. Then, local states of the two respected capsules are to be updated and, finally, three constraints are to be satisfied so that the rest of the application state remains unchanged.

```
pred migrate(c_src,c_dst:Capsule,d:DeployedComp,t,t':Time){
   c_src != c_dst
   #(c_dst.comps.t) < int[c_dst.comp_capacity.t]
   c_dst.comps.t' = c_dst.comps.t+d
   c_src.comps.t' = c_src.comps.t-d
   all capsule:Capsule|capsule.bindings.t'=capsule.bindings.t
   all capsule:Capsule-c_src-c_dst| capsule.comps.t'=capsule.comps.t
   all capsule:Capsule| capsule.comp_capacity.t' = capsule.comp_capacity.t
}
```

Above all those previous definitions, the RUNES Metamodel also enforces a couple of RUNES specific restrictions over the possible component configurations in order to safeguard that only semantically correct component reconfigurations are permitted. In the metamodel (see Figure 1 and Figure 2) those rules are expressed either via cardinality constraints or via OCL expressions. Therefore, the Alloy formalism must incorporate the corresponding definitions, too. Here only the most important elements of that constraint set are summarized.

- A Binding or a Component must be contained within at most one single Capsule

```
no disj capsule1,capsule2:Capsule|
 some (capsule1.bindings) & (capsule2.bindings)
no disj capsule1,capsule2:Capsule|
 some (capsule1.comps) & (capsule2.comps)
```

- Two Bindings of the same type must not be deployed if they share the same Receptacle.

```
no disj b1, b2:DeployedBinding| (b1.deploy = b2.deploy)
 and (b1.mapping.DeployedComp = b1.mapping.DeployedComp)
```

- There must not be such a Binding within a Capsule that has a connected Component which is not deployed in any of the Capsules

```
no deployedBinding:DeployedBinding|some t:Time|
 deployedBinding in Capsule.bindings.t and
 (deployedBinding.mapping.DeployedComp not in Capsule.comps.t
 or deployedBinding.mapping[DeployedComp] not in Capsule.comps.t)
```

E. Behavior metamodel

The internal dynamics of the components' functional behavior is modeled in Alloy by an explicitly specified Finite State Machine (FSM) that takes into account all changes of the internal state of vital components, the conditionality of state transitions and the necessary action semantics required when a new state has been entered. Our FSM definition in Alloy mirrors the formal mathematical model following the generic principle of semantic anchoring [13].

```
abstract sig State{}
abstract sig Transition{
   trans: State -> State
}{
   one trans
}
abstract sig StartState extends State{}
abstract sig StartTransition extends Transition{}
pred transition[d:DeployedComp,t,t':Time]{
   (d.fire.t).trans.State = d.current_state.t
   (d.fire.t).trans[State] = d.current_state.t'
}
abstract sig StateMachine{
   states: some State,
   startState: one StartState,
   transitions: some Transition,
   startTransition: one StartTransition,
}{
   no (states & startState)
   no (transitions & startTransition)
}
fact Traces{
   ...
   all t:Time-TO/last[],d:DeployedComp|let t'=TO/next[t]|
     some d.fire.t => (transition[d,t,t'])
   all t:Time|some DeployedComp.fire.t
}
```

To round up the section, Figure 3 depicts a concrete model that instantiates the above introduced RUNES Metamodel in Alloy. It shows a snapshot from a dynamicly evolving component configuration of a sensor network scenario where the components have been deployed over a cross shaped capsule topology - indicated by green arrows - within which the resource pools are also capacity limited. The mapping of the components and bindings onto the capsules in a particular instance of time is visualized by the brown and red arrows respectively.

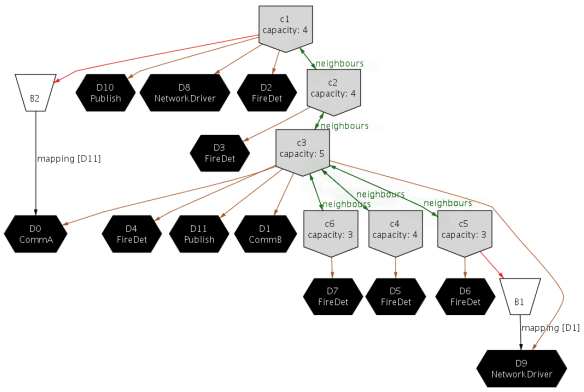

Figure 3. Scenario analysis snapshot

III. PROCESS

The RUNES application development process has a well defined five-layer architecture [3] that guides the application developer through the Scenario, the Application Modeling, the Platform, the Code Repository and the Running System stages. The presented Alloy based model verification approach builds on a first-order logic based formalism, which extends the RUNES development process. As Figure 4 shows, the extension has been realized by two additional model transformations that turn RUNES Component Models and corresponding RUNES Deployment Models into configuration scenarios that can be verified within the Alloy

 ISBN: 978-1-61208-130-4

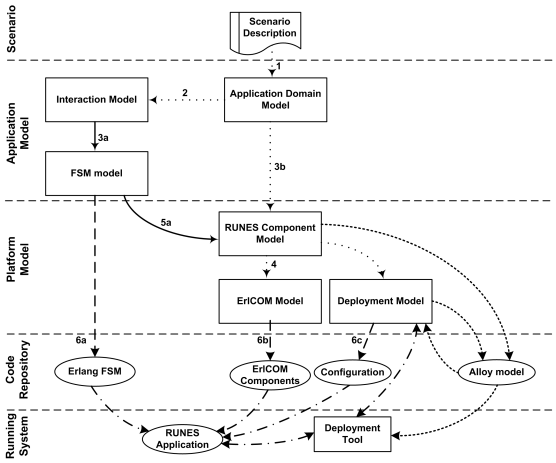

Figure 4. Software Development Process extended with Alloy verification

Analyzer. The model transformations produce configuration scenarios which contain both structural and behavioral specifications of the application. However, only those parts of the FSM action semantics are kept from the total dynamic behavior which directly relate to the internal control logic of the scenario. These parts precisely specify when and with which parameters the application invokes the runtime APIs provided by the RUNES middleware.

The verification of a particular scenario investigates the evolution of the application from the point of view of the component reconfigurations enabled by the RUNES middleware which are mainly restricted by the resource availability within the capsules along the time. The results of the verification provide input to the runtime autonomic control mechanisms that manage pre-calculated adaptive component reconfiguration. The approach is usually iterative and the convergence criteria are decided on a case-by-case basis.

IV. Metamodel-driven component management

Metamodel-driven component management is an interesting new way of generalizing policy-based network management [16] in such a way that the information model used by the network management infrastructure mirrors those software assets of the component based system that are produced by the model translators. In effect, the model based system design is kept intact and extended by elaborated action semantics. From the point of view of model-based application control, the most important element in the RUNES runtime architecture is the Deployment Tool, which establishes a soft real-time synchronization loop between the GME model repository and the running component application. The schematics of the Deployment Tool based reconfigurability is shown in Figure 4. The Deployment Tool, a protocol independent abstraction of GANA's Decision Making Element [17], first deploys the initial component configuration of the application then it constantly readapts the component configuration by listening to both application and middleware notifications and by continuously re-evaluating the configuration in hand. The core of the control logic is based on the verification results from previous Alloy analyzes. Moreover, it visualizes the actual component configuration of the system in a metamodel compliant view within GME and also takes indirect corrective actions by modifying the resource availability of the capsules via RUNES middleware API invocations. Currently, the control logic is not automatically generated from a batch of Alloy verifications; however our aim is to adopt the GANA [17] control meta-model and to populate it via an automatic model transformation directly from the instantiated RUNES metamodel in the Alloy verification phase. With the control logic properly established, the Deployment Tool is capable to function both as a re-active or a pro-active component reconfigurator as reported in [3], [18].

V. Simplified scenario example

In this section, a simplified example will demonstrate how Alloy helps the model verification. For the sake of easy comprehension, here, only a simplified configuration example has been chosen, which incorporates merely two capsules and 8 deployed components. Although this logic based approach, in general, is rather resource intensive, realistic scenarios by a magnitude larger in size are still possible to be analyzed successfully in this manner. Nevertheless, large reconfiguration setups must be optimized individually; therefore our current approach is, in virtue, semi-automatic.

In Figure 5, the Alloy representation of the functional configuration of the component system is depicted.

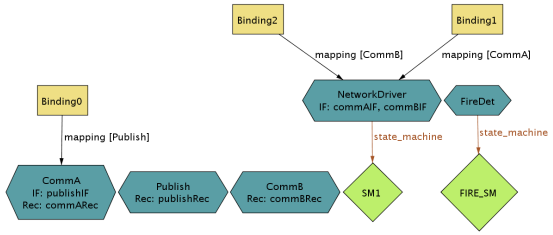

Figure 5. Functional configuration of the example system

The functional view of the investigated system contains five different component types, namely, the network related three components (NetworkDriver, CommA and CommB) and the two application specific components (Publish and FireDet). CommA and CommB implement two different communication paradigms relying on the functionality of the common NetworkDriver component through Binding1 and Binding2. The Publish component's main functionality is to broadcast different sensory measurement data towards the processing end points. The FireDet component is the control component which reconfigures the other components

 ISBN: 978-1-61208-130-4

whenever it has detected fire situation. The main goal of the reconfiguration is to keep the sensor system in operation even in case of extreme fire conditions. The reconfiguration is carried out by migrating the application functionalities to other capsules, which are located in the neighborhood. By decreasing the generic capacity parameter of the current capsule other capsules will not be able to immediately push back newly migrated components. Both the NetworkDriver and the FireDet components possess proper state machines which are represented by the diamonds in Figure 5.

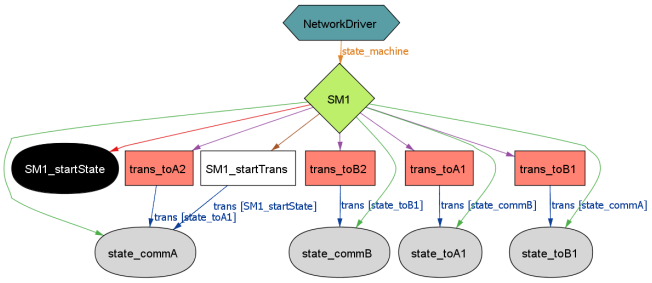

Figure 6. NetworkDriver component state machine

Figure 6 shows the state machine of the NetworkDriver component. The black ellipse shows the start state, while the white rectangle represents the transition from the start state to another state, which is called state_commA in this particular case. Via the transition from state_commA to state_commB, through a temporal state_toB1, the unbinding of component CommA from NetworkDriver and the binding of component CommB to NetworkDriver take place. This state change clearly represents the reconfiguration of the communication paradigm.

Figures 7–10 show an Alloy trace sequence. The resulting model is projected over Time in such a way that the relations rooted in Time are represented through a sequence of models. More precisely, one Time instance is connected to one particular Model snapshot.

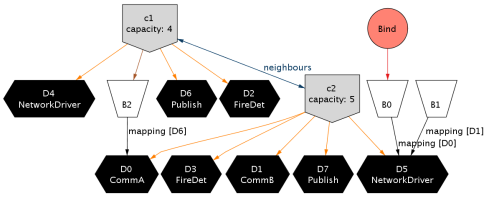

Figure 7. Component binding step

Figure 7 presents the first step of the sequence. When SM1_startTrans is activated the circle with the Bind tag points to the deployed binding B0. The deployed component D0 and D5 will be bound in the following step (see Figure 8).

In Figure 8 the first reconfiguration of the system can be seen. The FireDet component's state machine is activated,

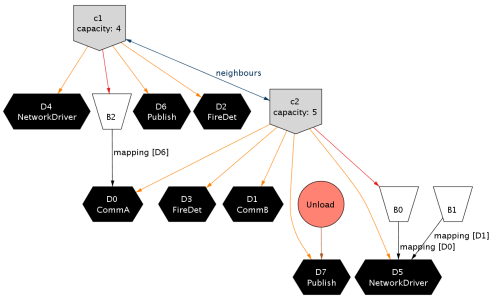

Figure 8. Component reconfiguration (unload) step

hence the migration of the application functionality has been started. Since the Publish component has been deployed to the neighboring capsule, the FireDet component, instead of migrating the marked component, is going to unload the Publish component from the second capsule. Furthermore, it will decrease the capacity of the capsule.

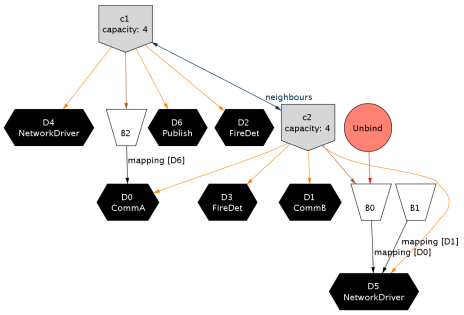

Figure 9. Component unbinding step

In Figure 9, the reconfiguration of the NetworkDriver component from CommA to CommB has started. In Figure 10, the second migration attempt is demonstrated. In this case, component CommA is migrating to the first capsule because this required functionality has not been deployed to that capsule so far.

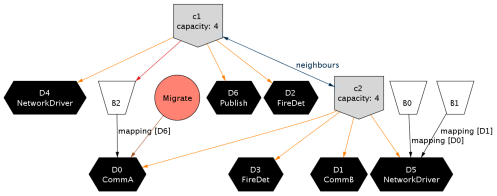

Figure 10. Component reconfiguration (migration) step

This simplified example indicates the way how a particular verification session takes place using the Alloy Analyzer. It helps generate configuration sequences which comply with application constraints. The current verification approach mainly focuses on the problem domain of component reconfigurability; thus, it assists the run-time control logic by

 ISBN: 978-1-61208-130-4

identifying situations with serious capacity limitations of the deployed capsules.

VI. Conclusion

This paper presents a new way of combining domain specific metamodeling techniques with first-order logic based metamodel verification so that model building could facilitate later run-time control mechanisms of the modeled system. We have introduced the semantical foundations of our approach and detailed its applicability in the case of reconfigurable component based sensor networks. A simplified example has been disseminated to illustrate the benefits of the approach. Our current work is to combine the RUNES meta-model and the GANA meta-model and to automate the generation of the adaptive control logic, based on the verification of the model based component configurations, to manage the deployed system. We are aware of the scalability issues of our approach, so further studies will be carried out in this regard. Moreover, the results of these studies will get incorporated, as best practices guidelines, into model translators that are supposed to produce the majority of the Alloy specifications. Ultimately, our aim is to create a generic framework which iteratively and interactively modifies and verifies the component model of sensor application scenarios and continuously indicates the most probable correct run-time configuration sequences thereof.

References

[1] "An architectural blueprint for autonomic computing." *Autonomic Computing*, IBM White Paper, June 2005.

[2] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis, "The runes middleware: A reconfigurable component-based approach to networked embedded systems," *Proc. of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05), Berlin, Germany*, September 2005.

[3] G. Batori, Z. Theisz, and D. Asztalos, "Domain specific modeling methodology for reconfigurable networked systems," *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, 2007.

[4] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, London, England, 2006.

[5] M. Taghdiri and D. Jackson, "A lightweight formal analysis of a multicast key man-agement scheme," *Formal Techniques for Networked and Distributed Systems (FORTE 2003)*, vol. 2767 of LNCS., pp. 240–256, 2003.

[6] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe, "An automated formal approach to managing dynamic reconfiguration," *21st IEEE International Conference on Automated Software Engineering (ASE 2006), Tokyo, Japan*, pp. 37–46, September 2006.

[7] D. Walsh, F. Bordeleau, and B. Selic, "A domain model for dynamic system reconfiguration," *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005)*, vol. 3713/2005, pp. 553–567, October 2005.

[8] E. G. Aydal, M. Utting, and J. Woodcock, "A comparison of state-based modelling tools for model validation," *Tools 2008*, June 2008.

[9] D. Jackson, "Alloy analyzer," *http://alloy.mit.edu/*, 2008.

[10] T. Erl, "Soa principles of service design," *Prentice Hall*, 2007.

[11] G. Batori, Z. Theisz, and D. Asztalos, "Robust reconfigurable erlang component system," *Erlang User Conference, Stockholm, Sweden*, 2005.

[12] G.Batori and D. Asztalos, "Using ttcn-3 for testing platform independent models," *TestCom 2005, Lecture Notes in Computer Science (LNCS) 3502*, May 2005.

[13] K. Chen, J. Sztipanovits, S. Abdelwahed, and E. Jackson, "Semantic anchoring with model transformations," *European Conference on Model Driven Architecture -Foundations and Applications (ECMDA-FA), Nuremberg, Germany*, November 2005.

[14] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The generic modeling environment," *In Proceedings of WISP'2001, Budapest, Hungary*, pp. 255–277, May 2001.

[15] I. H. Krueger and R. Mathew, "Component synthesis from service specifications," *In Proceedings of the Scenarios: Models, Transformations and Tools International Workshop, Dagstuhl Castle, Germany, Lecture Notes in Computer Science, Vol. 3466*, pp. 255–277, September 2003.

[16] L. Lymberopoulos, E. Lupu, and M. Sloman, "An adaptive policy-based framework for network services management," *Journal of Network and Systems Management*, vol. 11 , Issue 3, pp. 277 – 303, 2003.

[17] A. Prakash, Z. Theisz, and R. Chaparadza, "Formal methods for modeling, refining and verifying autonomic components of computer networks," *Advances in Autonomic Computing: Formal Engineering Methods for Nature-Inspired Computing Systems, Springer Transactions on Computational Science (TCS)*, Expected Publication: Winter 2010 (accepted).

[18] G. Batori, Z. Theisz, and D. Asztalos, "Configuration aware distributed system design in erlang," *Erlang User Conference, Stockholm, Sweden*, 2006.

 ISBN: 978-1-61208-130-4