Metamodel and Formal Logic based Methodology for Modeling, Refining and Verifying Reconfigurable Networked Component Systems

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

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

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 individual ad-hoc problem solutions and industrial best practices guidelines tuned to the particular domain specific modeling frameworks and methodologies. Targeting this deficiency, this paper disseminates an alternative, semi-formal methodology that incorporates a first-order logic based structural modeling language, Alloy, in the analysis of component deployment and reconfiguration. This novel approach could help to extend the limits of the generic domain specific metamodeling methodology that has been developed for creating Reconfigurable Ubiquitous Networked Embedded Systems.**

***Keywords*-Alloy; formal model semantics; metamodeling; dynamic component system; platform middleware; RUNES; Erlang; ErlCOM**

I. INTRODUCTION

Reconfigurable networked component systems provide a versatile implementation framework for highly distributed autonomic peer-to-peer applications targeting the domains of sensor networks and autonomous computing environments. The introduction of an effective, high-quality software development methodology, that speeds up the day-to-day tasks of application developers in such an inherently complex environment can be regarded as a rather valuable asset. In fact, the Reconfigurable Ubiquitous Network Embedded Systems (RUNES) IST project successfully completed this endeavor by providing a common distributed component-based platform architecture, on top of heterogeneous networks of computational nodes, and by establishing a corresponding model-based software development methodology and related framework implementation. Nevertheless, the practical building and later validation and verification of such networked component applications turned out to be quite an ambitious technical challenge, which almost always required detailed software engineering know-how that went beyond the usual precise understanding of the problem domain. Hence, we think that practical application development projects may enormously benefit from this beyond state-of-the-art domain specific modeling technique, which also includes some novel, formal logic based practical approaches. Although some of the early results have been reported in [1] the final validation of this methodology via practical application scenarios is still open for further investigation.

One of the major results of the RUNES [2] project was to establish a reflective distributed component-based multi-platform middleware architecture [3] for heterogeneous networks of computational nodes, including metamodel-based software development methodology [4] and graphical development framework. The RUNES metamodel provides all those relevant concepts that software developers must need to know in order to efficiently utilize the computational resources of a reflective distributed component-based environment. The complexity of these distributed reconfigurable component systems is due to the fact that the reflective components can be linked only by compatible provided-required interface pairs and their communication must be served either by these bound links, via pure message sending, or by a temporal storage of (meta-)data located in a distributed database. In the beginning of the RUNES project, only state-of-the art domain specific modeling techniques had been applied, however, later we had to realize that the usage of formal logic based language support, e.g. Alloy, could be taken advantage of in order to go beyond the traditional validation and verification approaches of state-of-the-art model based design methodologies. Therefore, we started experimenting with semi-automated domain specific model analysis techniques in Alloy that can be used to formally handle the evolution of some dynamic component behaviors in certain families of application domains for domain specific verification. As a contribution, this paper extends [1] by putting it into the context of our continuous-life-cycle model based development methodology and in this way also formalizes the relations between the metamodeling, platform and validation and verification part of it. Moreover, we firmly believe that through its practical applicability this methodology can contribute to the better automation of some modeling tasks by eliminating non-trivial dynamic errors or failure situations during the application design of reconfigurable component systems.

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

The paper is structured as follows: In Section I, the motivation for this research is introduced. In Section II, related works on Alloy usage for system verification and validation are presented. Then, Section III provides a background on the technical domain of reconfigurable networked component systems and the logical formalism of Alloy, which establishes the conceptual frame for the rest of the paper. Next, Section IV describes the methodology and its associated formal and semi-formal methods to designing, refining and verifying the components of reconfigurable systems. A case-study showcasing the usefulness of this methodology, focusing mostly on the application of Alloy in the case of a simplified scenario example, is presented in Section V. Finally, in Section VI, the conclusions and some insights into our future research are provided.

II. Related work

Distributed reconfigurable component systems are inherently complex to analyze, hence, the importance of formal description techniques in system design is well known in the scientific literature. In particular, in this paper, we restrict our work on the usage of Alloy [7] to target only practical scenarios where the model checking capability of a refuter seems to be powerful enough to assist the application developers. So, our methodology, though, reliant on a formal first order logic based description language, which is supported by a fully automated SAT solver based analyzer, it is still some way constrained when it comes to model complexity and scalability. However, we know from related scientific publications that similar formal description techniques of Alloy have been successfully applied to model various complex systems in a wide range of application domains for domain specific model verification purposes. It has been applied in [11] for the analysis of some critical correctness properties that should be satisfied by any secure multicast protocol. The idea of applying Alloy for component based system analysis was suggested also by Warren et al. [12]. That paper describes OpenRec, a framework, which comprises a reflective component model, and then its Alloy model is investigated in some details. This Alloy model served as a conceptual basis for our Alloy component model; however, our model is more detailed, which enables deeper analysis of system behavior. Moreover, [13] demonstrates another Alloy model that identifies various types of dynamic system reconfigurations. It provides a rather good categorization of various problems and corresponding solutions related to dynamic software evolution. Furthermore, Aydal et al. [14] found Alloy Analyzer one of the best analysis tools 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. This approach, in a nutshell, semi-automatically generates all relevant RUNES deployment configuration assets that will also be analyzed within Alloy. In effect, by analyzing a significant subset of frequently reoccurring configurations the boundary between valid and invalid component configurations can be better investigated against proper sets of model-based application and/or middleware feasibility constraints. The analysis results can be later reused for providing useful inputs to the run-time adaptive control logic in order to extend the model-based software development framework [4] with effective autonomicity.

In the rest of this paper, we will describe how this 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 [4].

III. Background

A. Networked Reconfigurable Dynamic Component System

The aim of any networked reconfigurable component systems is to hide the heterogeneity of the participating nodes from the view of the application. The RUNES architecture consists of a reflective reconfigurable component system and a corresponding Component Run-Time Kernel (CRTK). This means that the reflectivity of the CRTK manifests in the reifiability of all kernel elements via an explicit management interface, and the concepts of a component system lies in the heart of its implementation that complies with well-known component-based software engineering principles. In more details, the reflective components are linked together by their interfaces, they communicate via message sending and store their meta-data in a distributed database. Each computational node incorporates an instance of the CRTK, which provides the basic middleware APIs of component management. These architectural concepts were turned into an effective reference implementation, called ErlCOM [5], which runs on top of the Erlang/OTP distributed infrastructure [6]. ErlCOM being a full-fledged realization of the RUNES CRTK, the RUNES component system will now be described through ErlCOM terms.

A component is the basic unit of the system that corresponds to an active actor-like process, which contains some executable code and has a unique name that is registered in a global registry. The components are spread over caplet hierarchies, caplets being components themselves, in a pool of networked nodes. The root of the caplet hierarchy is called capsule, which is the main process entity of the node. The caplets' main purpose is to provide supervisory facilities for the maintenance of robustness and longevity of the whole component system. The supervisory decisions are taken according to a set of predefined constraints stored within a particular component framework. Examples of robust auto-configuration can be the reactivation of crashed components or the migration of a cluster of running components due

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

to e.g. load balancing. The main interaction between components is carried out by means of pure message passing through the bindings, which represent the behavioral policy on the communication channels. The bindings themselves are also components with special communication properties. Message passing is synchronous; messages can be intercepted both before entering the interfaces of the recipients and after the replies have been exited those same interfaces. The pre- and post-actions of the bindings constitute a list of additional transformations on individual messages. It is important to emphasize that by the introduction of the binding concept both concurrent code execution inside the components and individual message passing activities through those bindings can be reified and reasoned on via a reflective component configuration graph provided by the middleware. Bindings are created when a receptacle, that is, a required interface, of a particular component is to be bound to a provided interface of another component, provided that they have been found compatible. Finally, both the components and the bindings are facilitated with explicitly attached state information, which may also be associated with some additional metadata stored in a global repository, redundantly distributed over the meshed networked nodes.

The concepts and the specificities of the RUNES component system are specified on various levels of technical details and also from different perspectives. Firstly, the constituting concepts with their corresponding static and dynamic constraints are formalized within domain specific metamodels [4]. Secondly, the dynamics and fine-grained functional and operational details of the ErlCOM reference implementation of the RUNES CRTK have been specified both in Message Sequence Charts (MSC) and related Erlang source code snippets [5]. Finally, a conceptually though simplified, but semantically compatible formal logic based representation of core CRTK elements and operations have been defined in [1].

B. Alloy

For precise validation and verification of application models logic based tools provide exact, though sometimes theoretically complex and practically limited, answers to some of the most important configuration or dimensioning questions. Under validation we mean here the semantic compatibility of the designed system, only from the perspectives of configuration and dimensioning aspects, against the semi-formal and/or verbal specification of given use case scenarios. Verification has also a slightly limited scope in our interpretation since we rather rely on a refuter than a theorem prover in order to gain in practical applicability. Nevertheless, in the particular case of dynamic component systems deployed in the domains of sensor networks, we believe that the theoretical prowess and the practical applicability of some first order logic based techniques can be though efficiently merged for effective applicability. In this paper, our selected choice of formal logic description is based on Alloy [7], which is a textual modeling language relying on structured first-order relational logic with equality. Although other temporal logic based techniques such as Linear Temporal Logic (LTL) or Computational Tree Logic (CTL) constructs could have been applied, instead of Alloy's formalism, to our domain of investigation, we do think that Alloy's syntax lies closer to the spirits of current state-of-the-art programming languages and therefore it is way easier for the practical program developers to use it or understand generated Alloy expressions without having to delve into theoretically precise definitions of its constructs. However, by not directly relying on a temporal logic based model checker such as UPPAAL [8] or SPIN [9] we were, obviously, forced to recreate the temporal aspect of the evolution of component configurations as we reported in [1] and as it will be described more in details in IV-C of this paper. In general, Alloy's syntax is rather simple; a particular model in Alloy contains a set of signature definitions with fields, facts, functions and predicates. Each signature denotes a set of atoms, which represent the smallest building blocks of the language. Atoms are, per definition, immutable and uninterpreted. Each field must belong to a signature and represents a relation with some other signatures. Facts define constraints on other elements of the model. Functions serve as named containments of Alloy definitions and predicates are considered like parameterized constraints that can be invoked within facts, functions or other predicates. Alloy is supported by a fully automated constraint solver, called Alloy Analyzer [10], which can be used to verify model parameters by searching for either valid or invalid instances of the model. Model checking is achieved by automated translation of the model into a Boolean expression, which is analyzed by SAT solver plug-ins, which can be easily incorporated into Alloy Analyzer. Once an instance violating an assertion has been found within the defined scope of a particular analysis task, the result of the verification is declared as not valid. However, if no instance has been found, it is not, in any means, a proof that the assertion is valid, though in practical applications, it could be considered as such, though it still might be invalid within a larger scope. This non-monotonic behavior of the prover may be disturbing in theory, but it works quite well in practical cases since the most relevant errors with practical significance occur in small, though non-trivial sized models in Alloy. Thus, the selection of the proper scope is an important trade-off of Alloy modeling and it should be carried out as precisely as possible within the constraints of practicality.

IV. Methodology

A. Process

All kinds of professional software developments are usually accompanied by some development processes that

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

safeguard industrial scale applicability of the chosen technology. Although there are many well-established and widely-used model based software development approaches, e.g. Rational Unified Process, that significantly influenced our work, the ambition level of our process design aimed at covering all the stages of component based application development, including generative metamodeling technologies. The overview of the process stages are depicted in Figure 1. Figure 1 is layered into five stages; namely, Scenario, Application Model, Platform Model, Code Repository and Running System. The arrows of the non-iterative part of the process, connecting together the artifacts of the various stages, are labeled by sequence numbers in accordance to their timing. In this paper, we only briefly outline the process by mainly concentrating on the inter-work between the major elements of the stages.

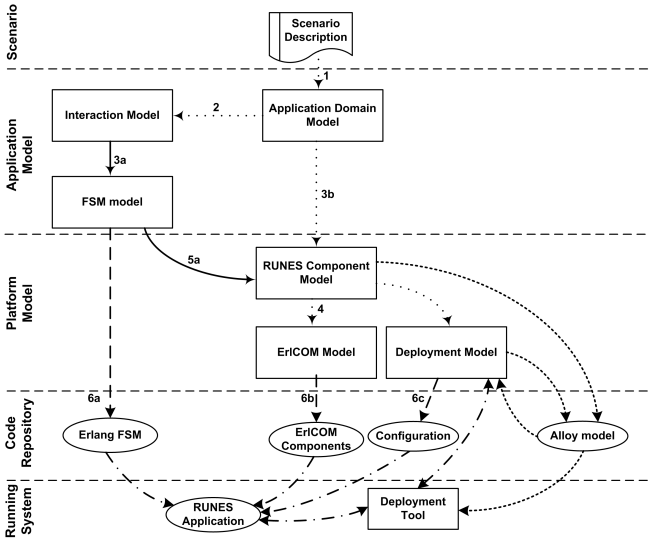

Figure 1. Software Development Process extended with Alloy verification

The Scenario evaluates and finalizes a set of scenario descriptions that establishes the exact scope of the application domain. Our experience gathered during the RUNES project showed clearly that reconfigurable component based applications can only be successfully developed if the application usage scenarios are detailed enough to enable non-trivial application modeling and quality analysis. Because realistic distributed applications involve intense interactions among application components both structural and interaction modeling are equally important. The Application Domain Model is created to cover the scenario in such a way that all use case details must be taken adequately into account and the stakeholders' roles have to be discovered, too.

The roles make up the basic elements of the interaction model, hence the dynamicity of the use cases must be translated into corresponding Message Sequence Charts (MSC). The Application Domain Model and the Interaction Model must be detailed enough so that quality investigations could be carried out in order to check the feasibility of the design. Moreover, this stage involves many creative decisions, so both arrow 1 and 2 in Figure 1 are dotted, this way showing that the activity is mainly carried out manually.

The Interaction Model is transformed into the Finite State Machines (FSM) Model and then a further translation maps it onto the RUNES Component Model. The solid arrow indicates that the translation is executed via graph transformations. The Application Domain Model usually requires creative refinements and only semi-automatically (see dotted line) can get translated onto the RUNES Component Model.

The Platform Model stage has been conceived to support total semantics elaboration, that is, the RUNES Component Model is extended by the semantics of the platform, the components and the FSMs. This step involves some manual coding in Erlang in order to produce a total executable specification of the application.

The final application model takes into consideration the distributed nature of the application; hence, the Deployment Model is populated. It entirely specifies the total component allocation of the application over the available nodes of the network.

The Code Repository is the stage which copes with source code management. The code production is fully automated, which is indicated by dashed lines. The Deployment Model is translated into an initial run-time configuration which is deployed over the available ErlCOM nodes. Any changes of the component configuration at run-time are managed by the Deployment Tool, which continuously updates the Deployment Model.

The Alloy based model verification step extends the standard operation of the Deployment Tool. It contains two additional model transformations; one that originates from the RUNES Component Model and another that takes a compatible RUNES Deployment Model and turns them into a configuration scenario that can be verified within Alloy Analyzer. The model transformations produce configuration scenarios, which include both the structural and the behavioral specifications of the application. However, only those parts of the FSM action semantics are kept from the total dynamic behavior that either directly relate to important control logic elements of the scenario or which belong to the operations provided by the underlying ErlCOM middleware. These steps simplify, though, precisely specify when and with which parameters the application invokes the CRTK of the RUNES middleware. Therefore, the verification of a particular scenario investigates mainly the evolution of the application from the point of view of its component reconfigurations that are allowed by the semantics of the ErlCOM middleware. More precisely, in our work we mostly targeted resource availability investigations over distributed capsules along the lifetime of the application. The results of this verification step provide useful input to the run-time autonomic control mechanisms either embedded inside the

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

application or defined as explicit rule-sets within autonomic extensions of the Deployment Tool, basically managing pre-calculated adaptive component reconfiguration. The verification step is rather iterative in nature, which is well supported by Alloy Analyzer, and thus the final convergence criteria are mostly decided on a case-by-case basis depending on the particular scenario.

B. RUNES Metamodel

1) Interaction Modeling: Large-scale networked systems can be efficiently comprehended as a large number of interacting services. By combining those services an entity is getting involved in the complete behavior specification for that entity is established. Therefore, the service concept is effectively based on the interaction patterns between the cooperating entities. The notion of a role describes the contribution of an entity within a given interaction pattern. In our work we followed a well-known service oriented approach [15], which maps a particular service specification onto a set of interconnected components, each of them having an internal FSM, and a corresponding pool of abstract communication channels. This methodology also advocates the use of state machine synthesis algorithms so that the scenarios can be quickly simulated and/or validated (see Figure 2).

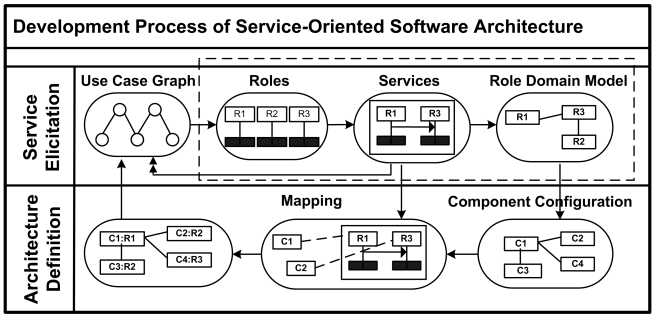

Figure 2. Service-based development

The generated state machines define the intended dynamic behavior of the specified system, thus they can be easily incorporated into our architectural design.

The state machine generation is carried out automatically and relies on two types of MSCs, the basic MSCs and the high level MSCs (HMSC). A basic MSC consists of a set of lines, each labeled by the name of the role and representing a certain unit of the behavior produced by that particular role. An HMSC is a graph whose nodes refer to other (H)MSCs. The semantics of an HMSC is obtained by following these operational paths and by composing the interaction patterns en route through the participating nodes. The output of the transformation is one FSM per role within the domain model; that is, the FSM implementing the respective role's contribution to the services it is associated with.

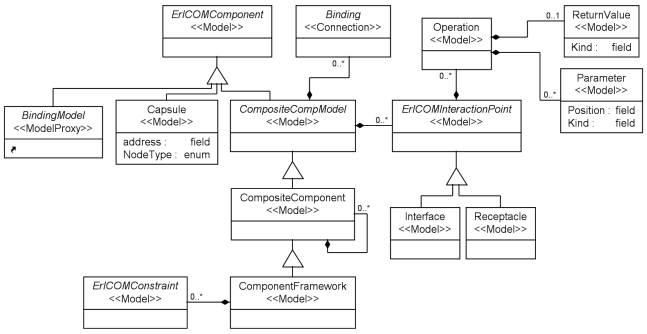

Figure 3. Functional metamodel

2) Functional Modeling: An outline of the component metamodel is illustrated in Figure 3. Components are encapsulated units of functionality and deployment, which interact with each other only via interfaces and receptacles. Interfaces are defined by a list of related operation signatures and associated data types. Components can provide multiple interfaces; embodying a clear separation of concerns (e.g. between base functionality and component management). Capsules and caplets are platform containers providing access to the run-time APIs. Bindings ensure consistent connection setup between a compatible interface and a receptacle. The component model itself is complemented by two other architecture elements: component frameworks and reflective extensions. Component frameworks (CF) are groupings of components with constraint guarantees to allow only "meaningful" component configurations. All entities of the metamodel (Component, Capsule, Interface, Receptacle, Binding, Component Framework) may store arbitrary <key,value> attributes, which contribute to a reflective layer facilitating universal discover at run-time. Component interactions can be intercepted at the bindings by pre- and post-actions to enable additional processing on the level of individual messages.

3) Behavior Modeling: The component behavior description is formalized in an abstract model of action semantics (see Figure 4). This Behavior Model is rather generic, but it though provides an explicit attribute for the specification of the modeled behavior within a particular implementation language. Those entities of the metamodel that may contain behavior descriptions are the Interface and the Component. A component model is translated onto target implementation languages by various model interpreters. In this way, the components can be created in various languages; however, they rely on the same modeling framework. A language specific model interpreter processes only those parts of a component model which contain relevant information for the desired target language environment. Therefore, the metamodel embodies various code snippets; the snippets are later woven together into executable component implementations by the related model interpreter. The most important parts

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

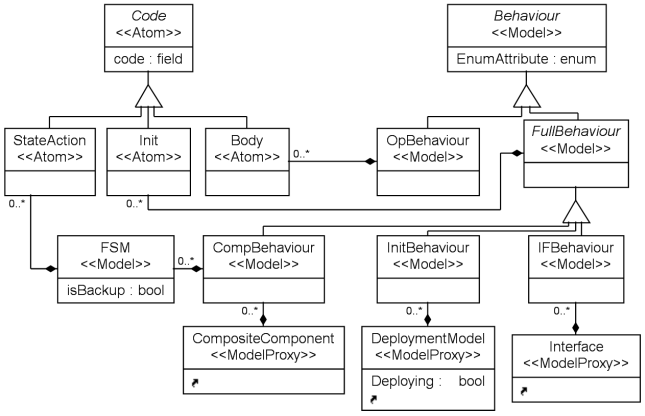

Figure 4. Behavior metamodel

of the code snippets are:

- Init - Initialization code for a component, an interface or the system.
- Body - Executable specification of the operation of an interface. The signature of the operation is defined in the model and automatically generated by the interpreter.
- StateAction - Specifies the semantics inside an FSM state. This action semantics is automatically injected into the corresponding connection point within the generated FSM Model.

4) Deployment Modeling: The complete synthesized platform specific application model contains both the structural configuration and the behavioral semantics of all the constituent components, including their interconnecting bindings and component framework constraints. That model represents the functional view of the application; however, it neither specifies how the application is deployed on the available networked nodes nor how it should start. Therefore, the deployment configuration must be modeled, too (see Figure 5).

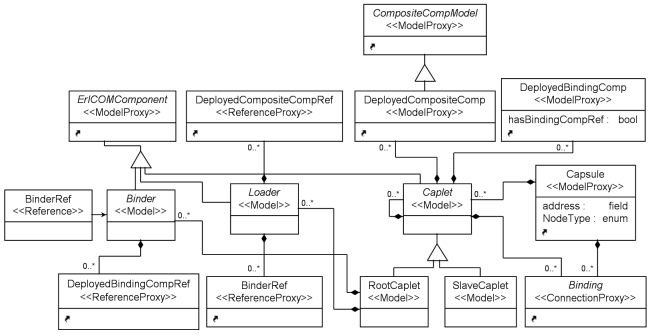

Figure 5. Deployment metamodel

The deployed component configuration, which contains the complete synthesized platform specific application model and the initial configuration of the components, is called the total synthesized platform specific distributed application model.

From the point of view of model based development, the most important element of the deployment infrastructure is the Deployment Tool, which establishes a soft real-time synchronization loop between the model repository and the running application. The schematics of the Deployment Tool based reconfigurability is shown in Figure 6.

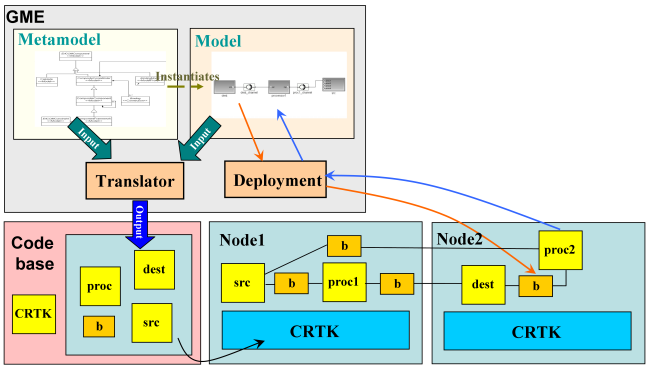

Figure 6. Deployment Tool based reconfigurability of run-time component application

The Deployment Tool analyzes the initial component configuration of the total synthesized platform specific application model and creates the needed ErlCOM elements by relying purely on the ErlCOM API. (Complete API semantics has been reported in [16]) After the initial deployment has been completed the application starts running and the ErlCOM CRTK continuously monitors all component reconfiguration and in case of observable component changes events are sent containing descriptive notifications to the Deployment Tool. The Deployment Tool keeps track of the actual component configuration of the running system by updating the total synthesized platform specific RUNES application model. Deployment Tool plug-ins can also execute policy based rules either re-actively or pro-actively. Any corrective changes on the modeled component configuration of the component application will be reflected by the run-time deployment.

C. RUNES Metamodel Verification with Alloy

1) Introduction: This section revisits the kernel part of metamodel, which defines the basic concepts of Interfaces, Receptacles, Components and Bindings, in order to formally represent those elements in a first order logic based formalism. Figure 7 illustrates that kernel part of the metamodel, including all the relevant relations and cardinalities. The associated OCL expressions are not visualized, though they play a significant role to establish a model-based rapid application development environment in the Generic Modeling Environment (GME) [17]. Mostly these OCL expressions

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

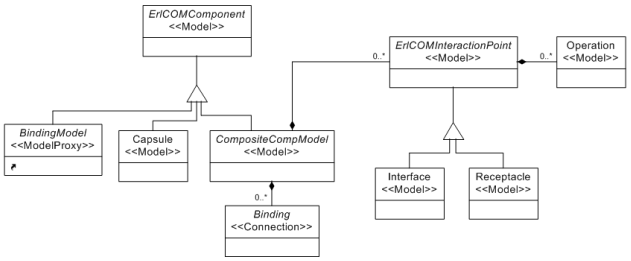

Figure 7. Kernel part of RUNES metamodel

interact with the generic component meta-model by further restricting the compatibility of component interconnection by creating an implicit subsumption hierarchy of bindable required-provided interfaces. Since a generic mapping of these OCL statements onto corresponding logical expressions in Alloy simply too complex and would go far beyond the scope of this paper, our current logic based formalism only relies on some selected elements of those OCL expressions and their mapping to Alloy has been ad-hoc and hand-crafted.

The generic aim of the approach is to to verify particular properties on some configuration sequences of certain modeled application scenarios via semantically anchored precise structural and behavioral formalism expressed in Alloy. The following sections describe the individual mappings between the various metamodeling concepts and their Alloy equivalents in adequate details.

2) Functional Model: 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 participating components take place exclusively via explicitly defined Interfaces and Receptacles. The dynamic behavior of the components are automatically generated from MSC and they are represented via concurrent FSM [15]. Intending to rewrite the above specification into Alloy, a generic RUNES Component is, hence, defined as a signature whose fields consist of at most one Finite State Machine and a set of Interfaces and Receptacles, respectively.

```
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-point of a inter-component connection, respectively.

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

A connection between compatible "provided" and "required" communication end-points is set up via Bindings. In fact, a Binding makes sure that connections between Interfaces and Receptacles are created consistently, according to compatible properties defined on the corresponding end-points. Hence, a definition of a Binding is also a signature in Alloy, however, it also contains some more fields; one for the Interface and another one for the Receptacle and finally a third one for a non-identical, component correct mapping connecting together the previous two fields. The connection constraint emanating from the "provided" and "required" characteristics of the end-points is attached to the Binding signature in the form of explicit logical restrictions.

```
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
}
```

Furthermore, a Receptacle must always represent a required set of operations that is a 'subset' of those operations which are provided by the Interface it intends to be connected to via the Binding. In RUNES application models, this requirement is specified by explicit operation signatures, including parameter lists either via OCL constraints or by the graphical representation of the metamodel. In the case of Alloy, this constraint semantics has been slightly simplified and only an abstract signature matching is enforced.

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

3) Deployment Model: Figure 8 revisits the most important deployment concepts of the RUNES Metamodel, which determine the runtime aspects of any component application. The key element is the Capsule, which represents the generic middleware container providing direct access to the functionalities of the runtime API of the CRTK. This set of functionalities incorporates also the robust fault management and recovery and the corresponding redundancy facilities. From the verification perspective, deploying a component into a capsule means that the capsule must be ensured to possess adequate resources made available for loading in components or bindings at any particular instance of time. The deployed components and bindings might be reorganized as time evolves, hence this temporal representation must take into account the explicit definition of time, too. This requirement can be easily satisfied by the introduction of Time into the formal representation of Capsules in Alloy. Hence, a Capsule is defined again as a signature, but this time it has also an explicit field standing for a time instance. The representation of the temporal evolution is not only

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

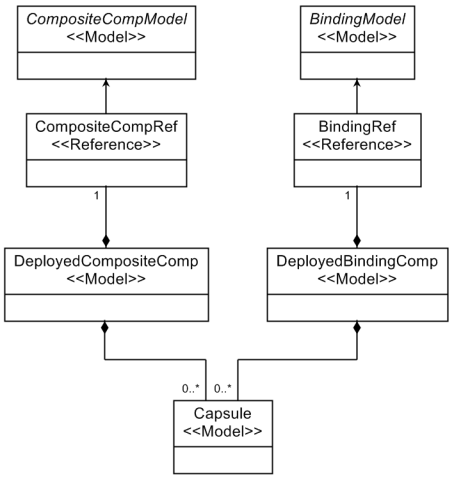

Figure 8. Deployment part of RUNES Metamodel

restricted to the deployed components or bindings, but it also incorporates a middleware related generic resource pool. This resource pool is again a semantic simplification, that abstractly represents the capacity of a capsule to contain either CRTK objects, such as components, or application specific elements that are usually stored in generic run-time repositories within capsules. Another semantic simplification is the time invariant representation of the capsule topology, though it may change in the case of a deployed application in real-time. Nevertheless, by setting the capacity of any particular capsule to zero one can easily simulate all kinds of run-time dynamic reconfigurability of a capsule hierarchy.

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

The formal specification of a deployed component must contain all those pieces of information that the active process aspect of the component's functionality requires, including the explicit definition of all state transitions in its FSM during the whole lifetime. In other words, structurally a deployed component has to be formally regarded as a dynamic instance of a component in accordance to its "ModelProxy" declaration in GME [18], as depicted in Figure 8. Considering the temporal aspect of its behavior, the state transitions are defined twofold in the signature of DeployedComp: on one hand, the fire mapping describes the fired transitions, one-by-one at a time; on the other hand, the field current_state tracks all state changes as time flies by.

```
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
}
```

Some additional constraints are also appended to the definition of DeployedComp. Firstly, it must be safeguarded that the deployed component honors the functional definition of its component description in such a way that its provided logical representation fully satisfies the "ModelProxy" declaration in GME. Secondly, the behavior of the FSMs is restricted to enable only one of them to fire a single transition at a particular instance of time. This is a sequencing constraint on causality of time evolution, which is a restriction globally applicable to the component application. Thirdly, letting a component have an FSM internally it must be deterministic in nature, hence, there is only one current state representing the total dynamic status of the component. The latter two conditions can be relaxed, but the detailed investigation of their consequences is still work in progress.

In contrast to the deployed components, a deployed binding does not have to declare its time evolution explicitly; nevertheless, its formal definition in Alloy contains a mapping field that anchors the DeployedBinding into two deployed components it is connecting together. Hence, the signature of DeployedBinding looks time independent, though it tracks time evolution, but indirectly. Due to the intimate linkage between the definition of a binding and the bound two components, the compatibility of their reliance must be ensured. This extra condition is made explicit through three additional logical constraints appended to the signature of DeployedBinding, stating the uniqueness and functional compatibility of the connection. In other words, if the functional compatibility of the participating components of a binding can be proven, then, also the deployment of such components is assumed to be valid. This formal set of extra Alloy expressions is homologue to their "ModelProxy" declaration in the GME metamodel.

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

Being our main motivation of applying Alloy for the verification of allowed component configurations, the deployed component applications must be also represented in a compatible Alloy formalism, that is, via a collection of capsules that are continuously tracking the temporal evolution of each of the deployed components and bindings. The Capsule having already been formally defined, the deployed component application is represented via its deployment configuration as a set of Capsules. Therefore, the formal definition of DeploymentConf is quite trivial. Furthermore, Alloy's trace statements help verify this time evolution of the deployed application as will be shown in Section V; thus, successful runs are easily and interactively visualized

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

for human inspection, too.

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

4) Middleware Model: The ErlCOM middleware API supports a complete set of component management operations such as [un]loading, [un]binding and migrating of components. These operations contain complex negotiation protocols among various elements of the ErlCOM CRTK and the deployed components, thus, the execution of a particular API invocation may require some time to complete its functionality. The operations usually modify only the local states of the distributed application and keep the rest of the application's state space unchanged. Obviously, these complex concurrent middleware activities must be considerably simplified so that a compatible logical formalism can be within reach of practical usability. Therefore, in general, all of the potentially concurrent atomic API operations are to be serialized in such a way that one and only one of them is allowed to be executed at one particular instance of time. As a good example showing this simplification approach, the Alloy definition of the 'migrate' operation will be explained in detail in the sequel. In the case of the remaining ErlCOM operations [5] similar techniques have been applied in order to translate them into corresponding Alloy expressions.

A component migration is carried out between two capsules by moving an already deployed component between two consecutive points of time. In effect, the migration itself can be conceptualized as a sequence of invocations of individual ErlCOM API operations: 'Create Component', 'Load Component', 'Update Component', 'Unload Component' and 'Destroy Component'. Obviously, the CRTK provides an optimized single API operation for completing the component migration in one single step; however, for the sake of explaining our approach of simplification the above sequence is considered to be valid. Since our formal Alloy representation is agnostic to the means by which the local state of a component is being migrated from one capsule into another, the operations of 'Load Component', 'Update Component' and 'Unload Component' can be either totally disregarded or taken into account in such a way that only the application relevant state space of the component is copied from time t to time t'. Hence, the only API invocations to be mapped into Alloy are 'Create Component' and 'Destroy Component'. Due to their analog treatment, let us examine only the operation 'Create Component'. The executable specification of the operation in Erlang is the following (see Figure 9 for corresponding MSC):

```
%create in CRTK
create(CapletName,InstanceName)->
   gen_server:call(global,CapletName , create,InstanceName ),
   insert_component(InstanceName,component,CapletName,CapletName).

%create in Caplet
create(InstanceName,Type)->
   CapsuleName = crtk:getOwner(crtk:getSelfName()),
   gen_server:call(global,CapsuleName, create,InstanceName),
   insert_component(InstanceName,Type).
```

```
%create in Capsule
create(InstanceName)->
   gen_server:start_link(global,ComponentName,
      e_EmptyComp, [InstanceName],[]).

%insert_component in Caplet
insert_component(InstanceName,Type)->
   ets:insert(get(componentTable),#componentcomponentName=InstanceName,
      componentData=#componentDatacomponentType=Type,state=created).

%insert_component in CRTK
insert_component(ComponentName,ComponentType,Owner,RegistryOwner) ->
   NodeName=node(),
   Fun = fun() ->
      mnesia:write(#componentcomponentName=ComponentName,
         componentType=ComponentType,owner=Owner,registryOwner=RegistryOwner,
         nodeName=NodeName)
   end,
   mnesia:transaction(Fun).
```

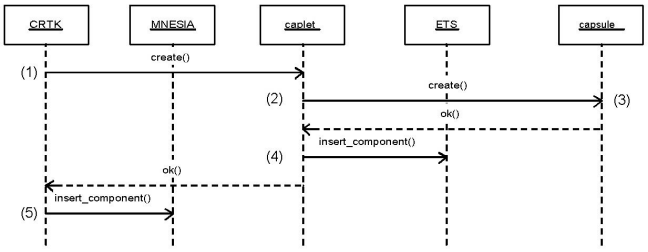

Figure 9. MSC of Create Component in ErlCOM

The message flow of a component creation is the following (see Figure 9): the CRTK first calls the create operatin on the caplet which forwards this request to the corresponding capsule. When the capsule responds OK the component related data will be stored into the caplet's local cache (ets). After the CRTK has received OK from the caplet, it registers the component data into the distributed Mnesia database.

Taking into account that the current Alloy specification of the Deployment Model (see Section IV-C3) only allows flat configurations of Capsules instead of a full hierarchy of Caplets with a leading root Capsule, steps 1 and 2 should be considered as a single combined activity. Moreover, steps 4 and 5 are only relevant to the internals of ErlCOM, therefore the first order logic based abstraction of 'Create Component' consists of one major task only, it being the addition of a new component into the receiving capsule. The operation of 'Destroy Component' can be handled similarly. Hence, in summary, the elements of the formal expression in Alloy of a migration operation are as follows: First the preconditions are checked if it is a real migration between two different capsules. Next, as capsules are abstracted to possess a generic capacity parameter, it is also checked if there are enough resources available in the receiving capsule. Then, the local states of the two respected capsules are updated, which is the homologue of the actual component migration in ErlCOM. Finally, three more constraints are to be satisfied in order to ensure that the rest of the application state remains unchanged. This restriction enforces our serialization concept of causality, which we intend to relax in our future research work.

```
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]
```

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
    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
}
```

Going beyond the dynamic aspects of ErlCOM CRTK, there are still some structural restrictions of the middleware left from the RUNES Metamodel. These enforce RUNES specific constraints over potential component configurations in order to safeguard the semantic correctness of component reconfigurations. In the metamodel (see Figure 7 and Figure 8) those rules are expressed either via cardinality constraints or by additional OCL statements. Consequently, their formal Alloy representation must be incorporated into our set of definitions, too. There are many such extra restrictions, though here we only introduce the most relevant elements of that constraint set.

- A Binding or a Component must be contained within at most one single Capsule. Until the binding or the component has not been deployed to or removed from a particular capsule its association with that capsule is non-existent. However, while it is deployed, one and only one capsule can contain it at any point of time.

```
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. This constraint enforces that connections of a particular type between two components, via a compatible Interface and Receptacle, cannot be shared at any point of time.

```
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. This constraint is critical since a binding can only connect together already deployed components at its related end-points. Unconnected or half-bound bindings are semantically incorrect since the bind and unbind operations of the ErlCOM API are both atomic in nature.

```
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)
```

5) Behavioral Model: The dynamic behavior of the component application is modeled via Finite State Machines (FSM) that are either automatically generated directly from the scenario MSCs or manually elaborated and added to selected components based on application specific requirements. Therefore, in essence, the internal dynamics of the components' functional behavior must be specified in Alloy by an explicit transcription of an FSM that specifies all changes in internal state of the component, including the preconditions of state transitions and the necessary action semantics required by the postconditions of the transition at entering the new state. Due to the complexity of practical applicability, only the vital components of the application are mapped onto their Alloy representation. Nevertheless, our FSM specification in Alloy is generic and mirrors the formal mathematical model following the abstract principle of semantic anchoring [19].

The formal Alloy definition of a Finite State Machine (FSM) relies on the signatures of State and Transition; the latter being specified as a mapping between two States. The initial state of the FSM is designated by the <StartState, StartTransition> pair. Since our verification approach targets the evolution of the FSMs in time a predicate named transition is introduced, which tracks the time instances and records the transitions being executed between every time t and t' inside the deployed component that has currently been chosen for letting its FSM fire.

```
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
}
```

The definition of the fact Traces puts the FSM into action, basically letting at most one transition fire at a particular point of time. The allowed firings are selected according to the defined transition rules within the FSMs; therefore, the deployed component application is totally FSM driven in our Alloy specification.

6) Example Model: Having all the elements of our Alloy formalism specifying the ErlCOM based, RUNES metamodel compatible models described in details, here the graphical visualization of such an example model is shown in Alloy Analyzer. Figure 10 depicts a model that represents a snapshot of a dynamically evolving component configuration of a sensor network scenario example. The components (black hexagons) have been deployed over a cross shaped capsule (gray pentagons) topology. The connections among the capsules of this topology are indicated by green arrows. The internal resources, here the maximum number of deployed components/bindings, of the capsules are pooled and limited in their capacity. The concrete mapping of the components and bindings (white rhombuses) onto the capsules, at a particular instance of time, is visualized by the brown and red arrows, respectively.

This figure shows only a particular snapshot of a dynamically evolving component application, therefore, for the

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

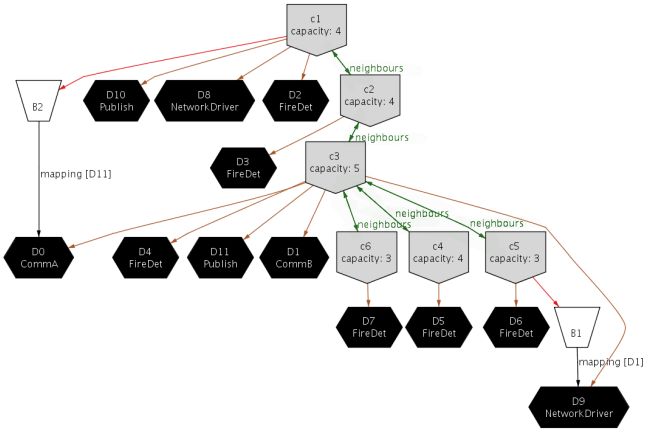

Figure 10. Scenario analysis snapshot

validation of an application scenario or the verification of a certain logical property full sequences of these snapshots must be analyzed. In the sequel, a simplified scenario model will be analyzed to demonstrate the Alloy driven verification step of our methodology in some detail.

V. Simplified Scenario Example

The scenario example that is used to showcase the usability of our proposed approach is based on the Fire in the Road Tunnel scenario [2] of the RUNES IST project. This simple scenario example is part of one of the RUNES demonstrators; hence, its elements have been extracted directly from a bigger component application in order to make it manageable for practical analyses in Alloy. For better understanding, some steps of the RUNES application development process (see Section IV-A) will also be shown in the case of this particular example. The relevant excerpt from the overview of the scenario story [2] is as follows:

"At the beginning of our story traffic is flowing normally in the road tunnel. Tunnel fires can be detected by the wired system that is part of the tunnel infrastructure. The fire sensors do, however, have the capability to operate wirelessly if required. An accident within the road tunnel has resulted in a fire. The fire is detected and is reported back to the TunnelControl Room. ... As a result of the fire the wired infrastructure is damaged and the link is lost between fire detection nodes. Using wireless communication, information from the fire detection nodes is still delivered to the Tunnel Control Room seamlessly. ... As the firemen move towards the fire the sensors reporting periodic data on external temperatures detect a rise in temperature and respond by increasing the frequency of reporting so that the EmergencyControl can assess the danger to the fire fighters. The fire becomes more severe. A node is lost..."

For the sake of being able to show dynamic behaviour modeling in Alloy, we are, first, focusing on the Interaction Modeling (see Section IV-B1). It is obvious to recognise that there is a Fire Detector service lying in the heart of the Fire in the Road Tunnel scenario. It was specified, within the RUNES project, via five MSCs, which are depicted in Figure 11.

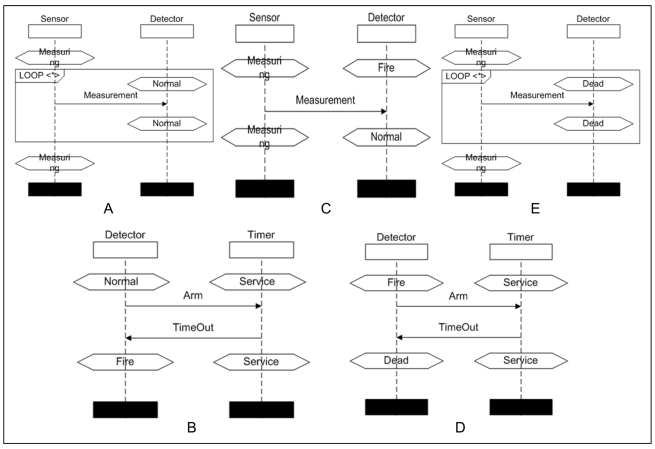

Figure 11. Message Sequence Charts of Fire Detector

Then, the MSCs are translated via a sequence of transformations [4], including a non-trivial graph transformation, into an equivalent FSM representation, which is shown in Figure 12.

Regarding its Functional (See Section IV-B2) and Deployment (See Section IV-B4) Modeling the scenario example has been simplified by having been selected only two capsules and 8 deployed components. In Figure 13, the Alloy representation of the functional configuration of the component system is depicted. The blue hexagons show the components, the beige rectangles represent the bindings and the green diamonds stand for the finite state machines.

This functional view contains, thus, five different components; namely, three network related components (NetworkDriver, CommA and CommB) and two application specific components (Publish and FireDet). The components CommA and CommB implement two different kinds of com-

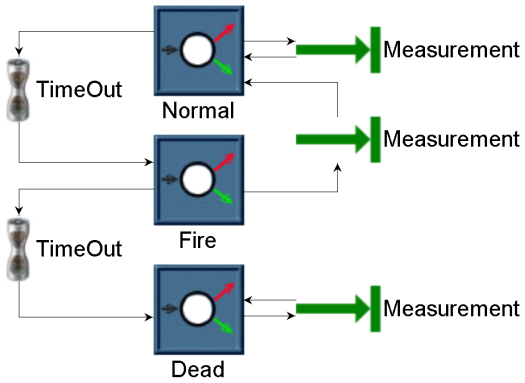

Figure 12. Platform Independent Behavior for Fire Detector

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

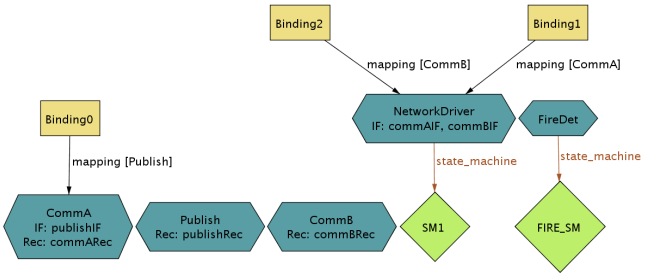

Figure 13. Functional configuration of the example system

munication paradigms, both of them relying on the shared functionalities of the common NetworkDriver component. Their connections are made available through Binding1 and Binding2. The main functionality of the Publish component is to broadcast different sensor measurement data towards the processing end points, such as the Tunnel Control Room. The FireDet component combines and simulates both the effects of spreading fire and the decisions that could have been taken by a real control component being responsible for reconfiguring the other components whenever a fire situation has been detected. Due to the complexity of its real-life homologue, FireDet in Alloy has a rather modified, adapted version of the original FSM that is associated with the Fire Detection service. Its control logic has been reduced to fit the trivial topology of the deployed components in this new functional setup. Nevertheless, the main goal of its functionality, which is to keep the sensor system in operation even in case of extreme fire conditions, has been left unchanged. Therefore, in the scenario example, the reconfiguration is carried out by letting the application components migrate to other capsules located in the neighborhood. In effect, the original states Fire and Dead have been combined and the substitute state causes a gradual loss in capsule capacity. By decreasing this generic capacity parameter of the capsule taken "by fire" the receiving capsule will not be able to immediately reinsert the recently migrated component; hence, ping-pong effects are eliminated.

Both NetworkDriver and FireDet possess proper state machines, which are represented by the green diamonds in Figure 13.

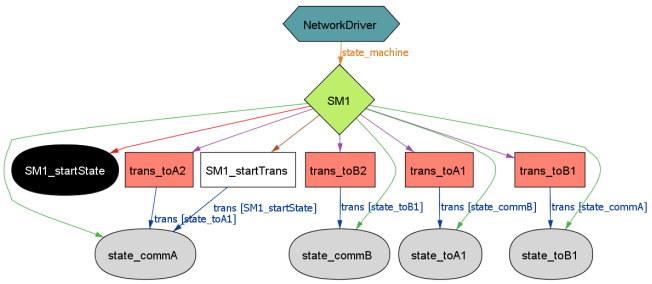

Figure 14. NetworkDriver state machine

Figure 14 shows the state machine of the NetworkDriver component. Its initial status is given by the start state, SM1_startState (black ellipse), and the initial transition, SM1_startTrans (white rectangle), leading from the start state to state_commA. (The states are represented by gray colored ellipses, while the transitions are shown via red rectangles.) Via a consecutive 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 will take place. This state transition sequence is a simplified version of the Behavioral model (see Section IV-B3) of the component and simulates the reconfiguration of the communication paradigms within the scenario example. The binding and the unbinding operations represent the invocations of the ErlCOM middleware API. (see Section IV-C4). The Alloy specification of NetworkDriver's FSM is as follows:

```
sig SM1 extends StateMachine{}
{
   startState = SM1_startState
   startTransaction = SM1_startTrans
   one SM1_state_commA
   one SM1_state_commB
   one SM1_state_toB1
   one SM1_state_toA1
   states = SM1_state_commA + SM1_state_commB +
       SM1_state_toB1+SM1_state_toA1
   one SM1_trans_toA1
   one SM1_trans_toA2
   one SM1_trans_toB1
   one SM1_trans_toB2
   transactions = SM1_trans_toA1 + SM1_trans_toA2 +
      SM1_trans_toB1 + SM1_trans_toB2
}

sig SM1_startState extends StartState{}

sig SM1_state_commA extends State{}{
   no t:Time-TO/last[]|
      let t' = TO/next[t]|this in getEndState[DeployedComp.fire.t] and
      not SM1_state_commA_action[t,t']
}

pred SM1_state_commA_action[t,t':Time]{
   some b:DeployedBinding|
      let d = getDeployedComp[SM1_state_commA,t]|
      b.mapping[DeployedComp] = d and
      ((b.mapping.DeployedComp).deploy = CommA) and
      bind[getCapsule[d,t],b,t,t']
}

sig SM1_state_commB extends State{}{
   no t:Time-TO/last[]|
      let t' = TO/next[t]|this in getEndState[DeployedComp.fire.t] and
      not SM1_state_commB_action[t,t']
}

pred SM1_state_commB_action[t,t':Time]{
   some b:DeployedBinding|
      let d = getDeployedComp[SM1_state_commB,t]|
      b.mapping[DeployedComp] = d and
      ((b.mapping.DeployedComp).deploy = CommB) and
      bind[getCapsule[d,t],b,t,t']
}

sig SM1_state_toB1 extends State{}{
   no t:Time-TO/last[]|
      let t' = TO/next[t]|this in getEndState[DeployedComp.fire.t] and
      not SM1_state_toB1_action[t,t']
}

pred SM1_state_toB1_action[t,t':Time]{
   some b:DeployedBinding|
      let d = getDeployedComp[SM1_state_toB1,t]|
      b.mapping[DeployedComp] = d and
      ((b.mapping.DeployedComp).deploy = CommA) and
      unbind[getCapsule[d,t],b,t,t']
}

sig SM1_state_toA1 extends State{}{
   no t:Time-TO/last[]|
      let t' = TO/next[t]|this in getEndState[DeployedComp.fire.t] and
      not SM1_state_toA1_action[t,t']
}

pred SM1_state_toA1_action[t,t':Time]{
   some b:DeployedBinding|
      let d = getDeployedComp[SM1_state_toA1,t]|
      b.mapping[DeployedComp] = d and
```

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
      ((b.mapping.DeployedComp).deploy = CommB) and
      unbind[getCapsule[d,t],b,t,t']
}

sig SM1_startTrans extends StartTransaction{} {
   trans[SM1_startState] = SM1_state_commA
}

sig SM1_trans_toB1 extends Transaction{} {
   trans.State = SM1_state_commA
   trans.State = SM1_state_toB1
   no t:Time|this in DeployedComp.fire.t and
      not SM1_trans_commA_pred[t]
}

pred SM1_trans_commA_pred[t:Time]{}

sig SM1_trans_toB2 extends Transaction{} {
   trans.State = SM1_state_toB1
   trans.State = SM1_state_commB
}

sig SM1_trans_toA1 extends Transaction{} {
   trans.State = SM1_state_commB
   trans.State = SM1_state_toA1
}

sig SM1_trans_toA2 extends Transaction{} {
   trans.State = SM1_state_toA1
   trans.State = SM1_state_commA
}
```

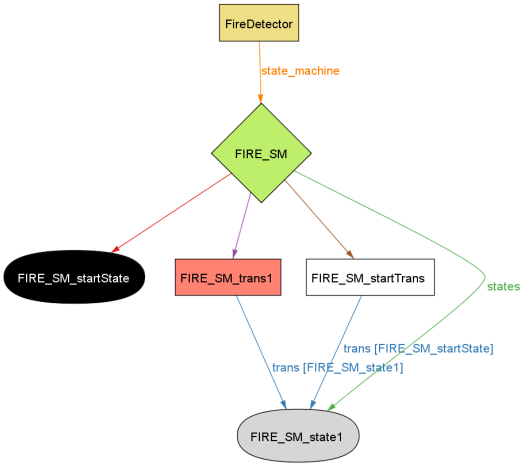

Figure 15. FireDet state machine

The Figure 15 shows the state machine of the FireDet component. As previously explained, this FSM is a reduced version of the original one possessing only two states. Its initial state, FIRE_SM_startState, represents the Normal state of the FSM associated with the Fire Detection service, while the state FIRE_SM_state1 stands for the combination of states Fire and Dead. (see Figure 12) The transition is one-way only and results in the decrease of capsule's capacity. The Alloy specification of FireDet's FSM is as follows:

```
sig FIRE_SM extends StateMachine{} {
   startState = FIRE_SM_startState
   one FIRE_SM_state1
   states = FIRE_SM_state1
   startTransaction = FIRE_SM_startTrans
   one FIRE_SM_trans1
   transactions = FIRE_SM_trans1
}

sig FIRE_SM_startState extends StartState{}

sig FIRE_SM_state1 extends State{}{
   all t:Time-TO/last[]|let t' = TO/next[t]|
      this in getEndState[DeployedComp.fire.t] =>
      FIRE_SM_state1_action[t,t']
}

pred FIRE_SM_state1_action[t,t':Time]{
   let c = getCapsule[getDeployedComp[FIRE_SM_state1,t],t]|
      decrease_capacity[c,t,t']
}

}

sig FIRE_SM_startTrans extends StartTransaction{} {
   trans[FIRE_SM_startState] = FIRE_SM_state1
}

sig FIRE_SM_trans1 extends Transaction{} {
   trans.State = FIRE_SM_state1
   trans.State = FIRE_SM_state1
}
```

In the sequel, a simple validation sequence of the above defined scenario example will be analyzed step-by-step. Figures 16–19 show the snapshots of an Alloy trace sequence. The model evolution is projected over Time in such a way that the relations of a model in different points of Time are represented through a sequence of consecutive models. Expressed it more precisely in Alloy parlance, it means that one Time instance is connected to one and only one particular Model snapshot.

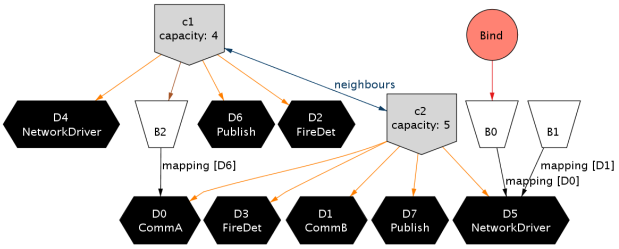

Figure 16. Component binding step

Figure 16 presents the first step of the sequence. When NetworkDriver_startTrans has been activated the red circle labeled by the Bind tag, which represents the invocation of the bind operation of the ErlCOM API, points to the deployed binding B0. The deployed component D5, in capsule c2, is going to be bound to D0 in the next step (see Figure 17).

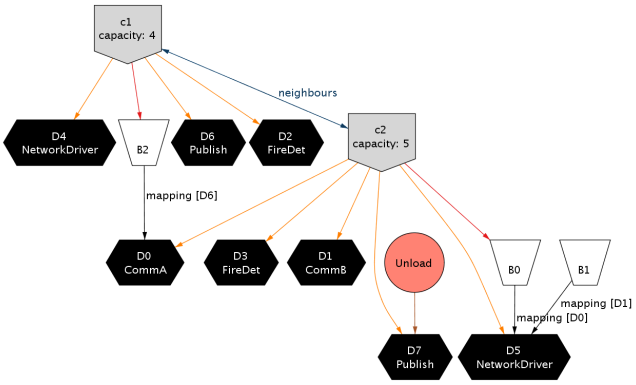

Figure 17. Component reconfiguration (unload) step

In Figure 17 the first reconfiguration of the system is to be seen. The FireDet component's state machine is activated; therefore, the migration of some application functionality has been started. FireDet selected the Publish component, in capsule c2, for migration; however, since another Publish

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

component had already been deployed to the neighboring capsule c1, the marked Publish component is going to be unloaded from capsule c2 instead of being migrated into capsule c1. Moreover, FireDet will also decrease the capacity of capsule c2.

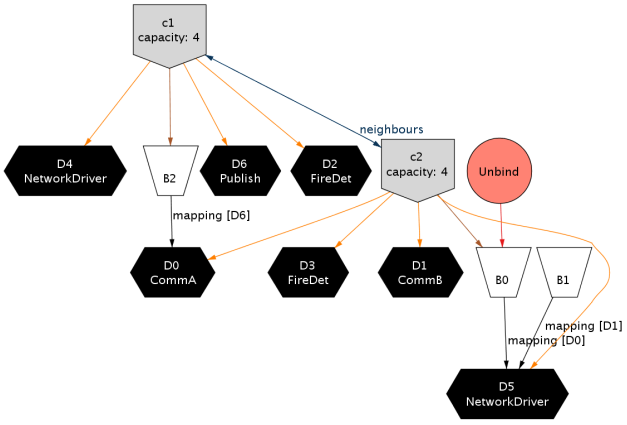

Figure 18. Component unbinding step

In Figure 18, the reconfiguration of the Network-Driver component has started changing from communication paradigm CommA to CommB. In Figure 19, the second migration attempt is demonstrated. In this case, component CommA is migrating to capsule c2 because this required functionality has not been deployed yet to that capsule so far.

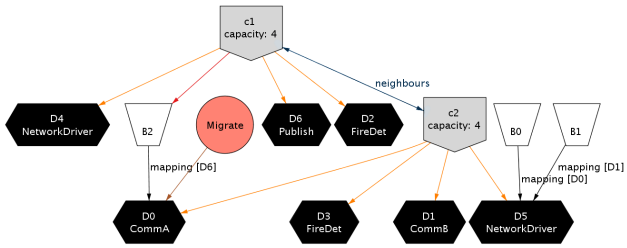

Figure 19. Component reconfiguration (migration) step

Although this example is a rather simplified one in nature, it indicates well the way how a particular validation or verification session may take place using Alloy Analyzer. Validation only generates a set of potential runs of a scenario, while verification also injects logical properties into the Alloy specification of the component application before it looks for counter-examples and shows them visually if found. In general, this approach helps enormously to analyze configuration sequences so that they both comply with some application constraints and avoid non-trivial pitfalls. The result of these analyses is later fed back to the control logic of the Deployment Tool (see Section IV-B4).

VI. Conclusion and Future Work

This paper has investigated a new way of combining domain specific metamodeling techniques with first order logic based model verification so that dynamic component applications could benefit from better quality reconfiguration mechanisms thanks to active scenario validation and verification. We have introduced the semantical foundations of our approach by describing the most relevant items of the RUNES metamodel, its development methodology and, most importantly, the first order logic based definition of those metamodel elements in Alloy representation. We have also illustrated the applicability of the approach in the case of reconfigurable component based sensor networks by a simplified scenario example that has been disseminated in detail. Our current work focuses on further extending the presented methodology by combining the assets of the RUNES and the GANA [20] metamodel in order to fully automate the generation of the adaptive control logic for autonomic component applications. So we are currently investigating the information extraction and feed-back of the results of Alloy based validation and verification of component model configurations so that we could explicitly manage the deployed system via multi-faceted control paradigms. Obviously, we are fully aware of the scalability issues of our current approach, so further studies will be carried out in this regard. Moreover, the results of these studies will be incorporated, as best practices guidelines, into future model translators, which are supposed to produce the major parts of the Alloy specifications and to evaluate the results of the analysis runs. 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 runtime configuration sequences thereof.

References

[1] Z. Theisz, G. Batori, and D. Asztalos, “Formal logic based configuration modeling and verification for dynamic component systems,” *Proceedings of MOPAS 2011*, 2011.

[2] K.-E. Arzén, A. Bicchi, G. Dini, S. Hailes, K. H. Johansson, J. Lygeros, and A. Tzes, “A component-based approach to the design of networked control systems,” *European Journal of Control*, 2007.

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

[4] 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.

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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

[6] J. Armstrong, "Making reliable distributed systems in the presence of software errors," *SICS Dissertation Series 34*, 2003.

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

[8] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04), LNCS 3185*, 2004.

[9] G. Holzmann and R. Joshi, "Model-driven software verification," *Proceedings of SPIN2004, Springer Verlag, LNCS 2989*, 2004.

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

[11] 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.

[12] 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.

[13] 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.

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

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

[17] 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.

[18] "GME documentation," *http://www.isis.vanderbilt.edu/Projects/gme*.

[19] 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.

[20] A. Prakash, Z. Theisz, and R. Chaparadza, "Formal methods for modeling, refining and verifying autonomic components of computer networks," *Springer Transactions on Computational Science (TCS) - Advances in Autonomic Computing: Formal Engineering Methods for Nature-Inspired Computing Systems in LNCS 7050*, pp. 1 – 48, 2012.

2012, © Copyright by authors, Published under agreement with IARIA - www.iaria.org