

Verification of Policy-Based Self-Managed Cell Interactions Using Alloy

Alberto Schaeffer-Filho, Emil Lupu, Morris Sloman, Susan Eisenbach
Department of Computing, Imperial College London
180 Queen's Gate, SW7 2AZ, London, England
Email: {aschaeff, e.c.lupu, m.sloman, sue}@doc.ic.ac.uk

Abstract

Self-Managed Cells (SMCs) define an infrastructure for building ubiquitous computing applications. An SMC consists of an autonomous administrative domain based on a policy-driven feedback control-loop. SMCs are able to interact with each other and compose with other SMCs to form larger autonomous components. In this paper we present a formal specification of an SMC's behaviour for the analysis and verification of its operation in collaborations of SMCs. These collaborations typically involve SMCs originated from different administrative authorities, and the definition of a formal model has helped us to verify the correctness of their operation when SMCs are composed or federated. The formal specification also enables a better characterisation of the integrity constraints that must be preserved during SMC operation.

1. Introduction

Management in complex pervasive environments cannot rely on human intervention, and systems must be *self-managing* with local decision making and feedback control to enable seamless adaptation. We have introduced the concept of a *Self-Managed Cell (SMC)* as an infrastructure for building ubiquitous computing applications [1]. An SMC implements a policy-driven feedback control-loop that determines which management and reconfiguration actions should be performed in response to events of interest such as device failures, context changes or changes of state in the SMC's resources. Autonomous SMCs must be able to interact with each other in complex ways, federate or compose into larger structures. Systems such as *body-area networks* for monitoring a patient's health must be autonomous and continuously adapt to changes in their environment or in their usage requirements. These may comprise "*smart sensors*" and complex diagnosis devices that are SMCs in their own right. Body-area network SMCs may interact with a number of other *peer* SMCs such as the SMC running on the PDA of a nurse or a doctor, or the SMC controlling the room in which the wearer is present. SMC interactions comprise the invocation of actions from one SMC on another but also exchanges of events and policies between SMCs.

In this paper we present a formal specification of the overall SMC behaviour and its analysis in collaborations across SMCs. In such collaborations, consistent policy deployment is crucial, as often SMCs form autonomous administrative domains. When these SMCs are composed or federated, inconsistencies may prevent them from operating as originally expected. The definition of a formal model assists in the design of SMC collaborations and allows us to verify the correctness of anticipated SMC interactions before these interactions are implemented or deployed in physical devices (e.g. PDAs, mobile phones, sensors). We show how our formal model is being used to automatically type-check policies (matching the events and operations specified in a policy to the functionality provided by the actual SMCs involved in the execution of that policy) and verify the consistency of collaborations, allowing us to identify conflicting or inconsistent policy deployment across distributed SMCs. The formal model also enables us to better characterise the foundations and elementary integrity constraints which are related to SMC management.

We chose the *Alloy Analyzer*¹ as the platform for this formal specification analysis. *Alloy* is a modelling language based on first-order logic. It is used for declaratively expressing complex structural constraints and behavior in a software system. Its analyzer can automatically check models for correctness, generate instances of model invariants, execute guided simulations involving a number of operations defined in the model and check user-specified properties of a model.

Defining a formal specification for the behaviour of Self-Managed Cells in *Alloy* allows us to: (1) formally capture the static and dynamic aspects of the structure and behaviour of the SMC interactions; (2) automatically verify the consistency of SMC collaborations and generate valid configurations – or generate counter-examples of invalid configurations – by using the Alloy analyser; (3) simulate SMC behaviour in complex interactions thus increasing our confidence in the suitability and correctness of a given model.

This paper is structured as follows: Section 2 presents an overview of the SMC architecture. Section 3 describes in some detail our formalisation of Self-Managed Cells, and Section 4 shows how this model can be used to automatically type-check and verify the consistency of collaborations of

1. <http://alloy.mit.edu>

Table 1. Examples of analysis properties of SMC interactions in Alloy.

	Consistent interaction establishment	Consistent policy deployment
Specification of model invariants (fail-safe defaults)	Any role must have either zero or one interface assigned to it.	Any active policy must have both its subject and target roles assigned to some interface (in the domain of the SMC enforcing the policy).
Specification of operations with valid state transitions	Departure of an SMC from the interaction causes the assignments involving the provided interfaces (and required roles) of that SMC to be removed.	A policy can only be activated if it was loaded before.
Specification of simulations describing part of an interaction	A sequence of predicates describing the discovery of an SMC, the assignment of an interface provided by this SMC to a role required by a second SMC, and the departure of the latter.	Idem plus the loading and activation of specific obligation policies among the two SMCs, if they have matching authorisation policies allowing the execution of the obligation policies.

SMCs. Section 5 outlines some related work, and Section 6 presents our concluding remarks.

2. Self-Managed Cells and their Interactions

An SMC forms an autonomous administrative domain that consists of hardware components such as physiological sensors, mobile phones, PDAs and computers, as well as software services and components within those devices. A typical set-up representing a patient's *body-area network* we use for healthcare monitoring comprises a Gumstix² device hosting management services that control several sensors (e.g., heart-rate, temperature, acceleration) hosted on BSNs (Body Sensor Nodes)³ as well as other devices such as diagnostic devices hosted on PDAs or other Gumstix. Communication with BSN nodes typically occurs through IEEE 802.15.4 radio links while communication between Gumstix devices or with PDAs occurs through Bluetooth or Wi-Fi.

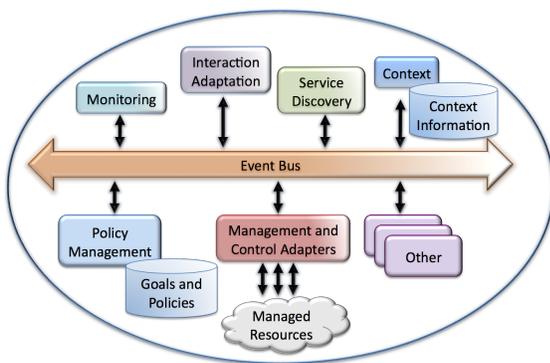


Figure 1. Self-Managed Cell architecture.

An SMC comprises a dynamic set of management services integrated through a common publish/subscribe *event bus* (Fig. 1). This de-couples services, as event publishers

do not require prior knowledge of the recipients when sending a message, and permits adding new services to the SMC without disrupting the behaviour of existing ones. The SMC relies on a *policy service*, which caters for two types of policies: *obligation policies* that define the adaptive actions that must be performed in response to events, and *authorisation policies* that specify which actions are permitted on which resources and services. Policies can be dynamically added, removed, enabled and disabled to change the behaviour of an SMC without interrupting its functioning. Our implementation of the policy service is based on the Ponder2⁴ system. Finally, a *discovery service* is used to detect new devices in the vicinity of the SMC, such as sensors and other SMCs, and is responsible for managing the SMC's membership, to distinguish transient failures from permanent departures from the SMC (e.g., device out of range or switched off). A more detailed description of the SMC architecture can be found in [1].

To realise larger applications, however, autonomous SMCs must be able to interact with each other in complex ways. We use the concept of *roles* to facilitate these interactions, where *roles* are placeholders for remote SMCs discovered at run-time. Roles can be thought of as typed-domains in a hierarchical *domain structure* that each SMC maintains to manage its resources. A remote SMC is assigned to a role in another SMC if it fulfills the requirements for that role i.e., if it provides the operations required by that domain. Once an SMC is assigned to a role, policies previously associated with a role (i.e., having that role as subject or target) will then apply to SMCs assigned to it. This allows us to specify policies governing the interactions with an SMC before that SMC is discovered. The implementation of SMC discovery and role assignment are described in [2].

Recently, we advocated the use of *architectural styles* for systematically specifying collaborations among *Self-Managed Cells* by composing patterns of interaction as design elements of a collaboration [3]. The use of architectural styles provides a better understanding of the

2. <http://www.gumstix.com>
 3. <http://vip.doc.ic.ac.uk/bsn/>

4. <http://www.ponder2.net>

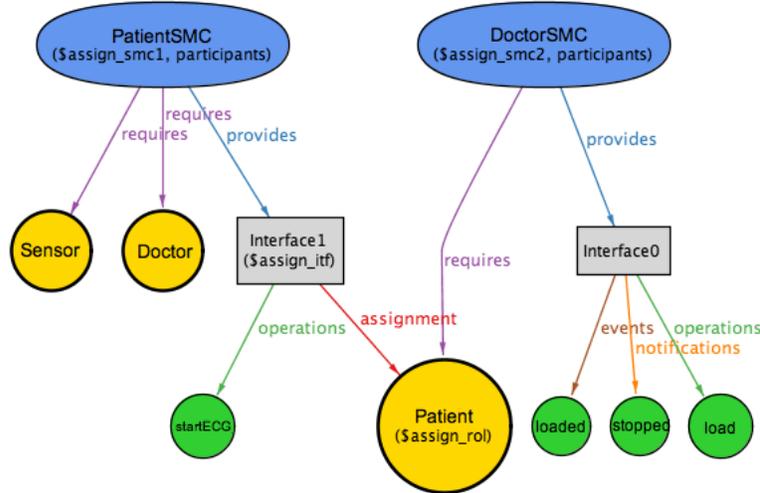


Figure 2. Alloy graphical representation of a simple interaction between SMCs.

interactions and promotes the reuse of common abstractions. Architectural styles are used to describe management relationships between SMCs and are similar in intent to software design patterns [4], in the sense that they provide a set of standard solutions for recurring problems. We distinguish between three main types of architectural styles: *structural styles* specify how SMCs are organised and structured and address issues such as interface mediation, filtering, encapsulation and SMC visibility; *task-allocation styles* specify control flow and policy loading strategies between the elements in a structural relationship; and *communication styles* specify information flow and event-forwarding behaviour between SMCs. In modelling complex SMC interactions we borrowed some ideas from software architecture-based approaches, which typically separate computation (*components*) from interactions (*connectors*) [5], [6], [7], [8]. However, components and connectors are generic abstractions that mostly deal with the structural bindings of software components. We want to represent higher-level relationships between components, catering for the adaptive behaviour of SMCs as expressed in policies and roles.

3. Formal Specification of Self-Managed Cells

Alloy [9] is a declarative modelling language used for expressing complex structural constraints and behavior in a software system. Models written in Alloy can be automatically checked for correctness using its analyzer. A visualizer can also be used to display example (or counter-example) structures graphically. The figures in this and in the next section were generated by this visualizer (with small hand edits of names to make the visualisation easier). Alloy performs a finite scope check, i.e. analysis in Alloy is performed over restricted scopes on the number of objects to be used, which is defined by the user (the user-specified scope makes the

problem finite and thus reducible to a boolean formula). This is based on the *small scope hypothesis* [10], which is, that for any flawed design a counter-example should be found by an exhaustive search within a comparatively small, bounded scope. [10].

Our Alloy specifications comprise: specifications of model invariants (i.e. fail-safe defaults), which define the integrity constraints; specifications of operations with legal state transitions, which describe the effect of a behaviour in SMC interactions; and simulations, which describe parts of a particular interaction involving a number of operations (i.e. a sequence of state transitions combined in a particular order). These can be used in the verification of both consistent interaction establishment between SMCs and consistent policy deployment. Table 1 presents examples of how we are using Alloy in the formalisation of SMC interactions.

Defining a formal specification of our interaction model in Alloy allows us to: (1) formally capture the static and dynamic aspects of the structure (through *signatures*) and behaviour (through *predicates*) of the SMC interactions; (2) automatically verify the consistency of given SMC collaborations and generate valid configurations – or generate counter-examples showing not valid configurations; (3) perform guided simulations through complex interactions and increasing our confidence on the correctness of a given collaboration.

The main component in our model is the SMC, represented by the signature *SelfManagedCell* below.

```

abstract sig SelfManagedCell
{
  provides: some Interface,
  requires: some Role,
  obligations: set Obligation,
  authorisations: set Authorisation
}

```

```

pred aGuidedSimulation [disj conf1,conf2,conf3,conf4: Configuration,
                        patientSMC,doctorSMC: SelfManagedCell]
{
  // discovery
  discover[conf1,conf2,patientSMC]

  // assignment
  one itf: patientSMC.provides,rol: doctorSMC.requires { assign[conf2,conf3,patientSMC,itf,
                                                                doctorSMC,rol]
                                                    }

  // departure
  departure[conf3,conf4,doctorSMC]
}

```

Figure 3. A guided simulation consisting of four steps and three predicates (discovery, assignment, departure).

In the declaration of a signature body we can define a number of relations, which can be thought of as fields of an object in the OO paradigm. The *SelfManagedCell* signature specifies four relations. The first two, *provides* and *requires*, define respectively which *interfaces* an SMC is able to offer to remote SMCs and which *roles* an SMC requires to be fulfilled (by remote SMCs). The other two relations, *obligations* and *authorisations*, define the policies an SMC is enforcing (which will be discussed shortly).

SelfManagedCell is an **abstract** signature, meaning it can be extended to define a specialized component in our model (e.g. *DoctorSMC*, *PatientSMC*, *SensorSMC* would all extend the base signature *SelfManagedCell*).

An *Interface* defines the *operations*, *events* and *notifications* supported by that interface, as described in [2]. It is defined by the following signature:

```

sig Interface
{
  operations: set Operation,
  events: set Event,
  notifications: set Notification
}

```

A *Role* behaves as a placeholder for remote SMCs, which can have their interfaces assigned to it. The complete model is too large to be presented here but is available online⁵.

Figure 2 shows an example of a valid configuration of interacting SMCs automatically generated from our Alloy specification. In this example, *PatientSMC* requires two roles (*Sensor* and *Doctor*) and provides one interface (*Interface1* – which supports the operation *startECG*). Similarly, *DoctorSMC* requires one role (*Patient*) and provides one interface (*Interface0* – which supports the event *loaded*, the notification *stopped* and the operation *load*). When the two SMCs discover each other, e.g., when the Doctor discovers the Patient, *Interface1* provided by *PatientSMC* is assigned to the *Patient* role required by *DoctorSMC*.

The behaviour of an Alloy model is defined through a set of *predicates*, which in our case represent the elementary behaviour of SMCs, such as the discovery and departure of SMCs, and assignment and unassignment of SMC interfaces to/from SMC roles. These operations can be easily modelled in Alloy, where the occurrence of an operation can be expressed by a change from a state *S* to a state *S'* that has a different set of properties. Thus we can model the changes that happen when an SMC is discovered, when an SMC departs, when an SMC is assigned to a role, and so on. This approach also permits to uncover omissions or implicit assumptions in informal models. For example, when an SMC *departs* from an interaction, the current assignments must be changed in a way we didn't clearly realise before: the new set of assignments must exclude not only the assignments involving the *interfaces provided* by the departing SMC, but also the assignments of other participants interfaces to the *roles required* by the departing SMC. The specification of the operation in the formal model helped us to precisely characterise this behaviour.

We can combine an arbitrary number of operations in a *guided simulation*, showing how the system evolves from a state *S* to a state *S'*, then from *S'* to *S''*, then from *S''* to *S'''*, etc, each time an operation is executed (e.g. after a departure, available SMCs must be reassigned to the roles previously fulfilled by the departing SMC). This allows us to precisely characterise the properties of each step of a dynamic collaboration that changes over time, as new SMCs may be discovered and interacting SMCs may depart frequently, consequently causing the number of assignments and active policies to change as well.

SMCs enforce two types of policies: *obligations*, which cater for the adaptive behaviour of SMCs, and specify what management actions (or *operations*) *subjects* must perform on *targets* in response to events; and *authorisations*, which are access control rules that specify what actions *subjects* are permitted (positive authorisation) or forbidden (negative authorisation) to perform on a *target*. The subjects and the targets are roles within the context of the SMC in which the

5. <http://www.doc.ic.ac.uk/~aschaeff/alloy/policy-model.zip>

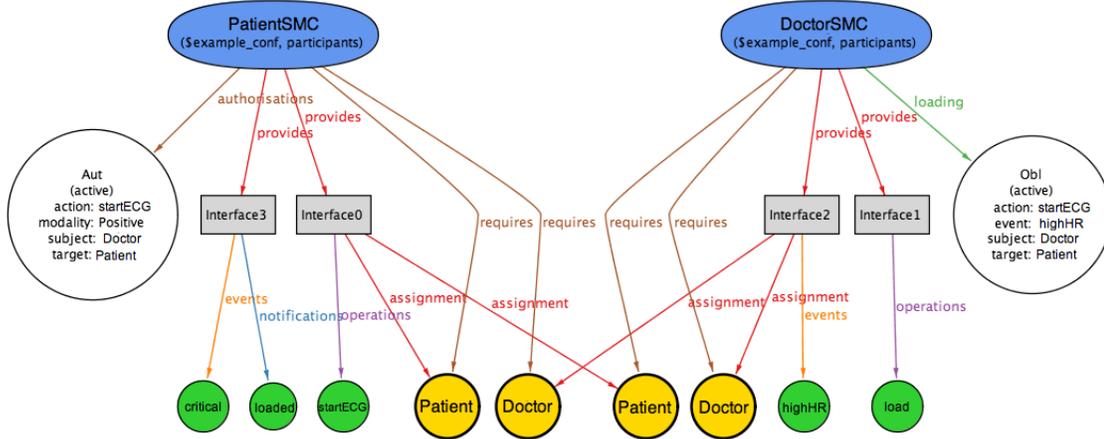


Figure 4. A valid policy configuration between SMCs (all active obligation policies have a corresponding active authorisation policy).

policy is specified.

The *ConcreteObligation* signature extends *Obligation* and defines the subject and target roles, and the event that triggers the policy and action to be invoked.

```
sig ConcreteObligation extends Obligation
{
  subject: one Role,
  event: one Event,
  action: one Operation,
  target: one Role
}
```

Similarly, the signature *ConcreteAuthorisation* defines a subject role, a target role, an action and the modality of the policy (which can be either positive or negative).

```
sig ConcreteAuthorisation extends Authorisation
{
  modality: one Modality,
  subject: one Role,
  action: one Operation,
  target: one Role
}
```

We defined additional predicates that cater for the loading and unloading of policies, and their activation and deactivation. As described above, these are used to show how the system evolves from a state S to a state S' when new policies are loaded, activated, or de-activated and removed.

We encode the notion of “state” of an interaction in an additional signature, named *Configuration*. A *Configuration* represents the interaction between SMCs at a given time point, and the various actions and exchanges performed in the interaction are thus represented as transitions between configurations. The “state” of an interaction is defined by the current set of *participants* in this interaction, the

assignments of participants (represented by their provided interfaces) to roles and the policies that are exchanged (*loaded*) between the SMCs. As shown in [1] collaborations between SMCs frequently require exchanges of policies between the SMCs regarding the duties of each party in the interaction. Such groups of policies exchanged are also termed missions. Additionally, the state of the interaction also depends on the policies that are active in each of the participating SMCs.

```
sig Configuration
{
  participants: some SelfManagedCell,
  assignment: Interface lone → Role
  loading: SelfManagedCell →
    (ConcreteObligation
     + ConcreteAuthorisation),
  active: set (ConcreteObligation
              + ConcreteAuthorisation)
} {
  active in (participants.obligations
            + participants.authorisations
            + participants.loading)
}
```

Figure 3 shows an example of how the evolution of an interaction can be represented through *guided simulations*. This simple example consists of four steps, and it shows how an interaction passes from the state *conf1* to *conf2* after *patientSMC* is discovered; then how the interaction passes from the state *conf2* to *conf3* after an interface provided by *patientSMC* is assigned to a role required by *doctorSMC*; and finally how the interaction passes from the state *conf3* to *conf4* after *doctorSMC* departs from the interaction. By using the visualiser tool we can clearly understand how the model of this specific interaction behaves according to the execution of the operations.

```

all conf: Configuration, smc: conf.participants, obl: (conf.active & ConcreteObligation) {
  ((obl in smc.obligations) or ((smc → obl) in conf.loading))
  ⇒ (obl.action in ((obl.target).~(conf.assignment)).operations)
    and (obl.event in (((obl.subject).~(conf.assignment)).events
      + ((obl.subject).~(conf.assignment)).notifications))
}

```

Figure 5. Type-checking restriction for policy activation: for all obligation policies, if the policy is active and an SMC has this policy, then the action of this policy is provided by the interface assigned to the target role and the event that triggers this policy is defined either as an event or as a notification in the interface assigned to the subject role.

In the next section we show how this model is being used to automatically type-check and verify the consistency of SMC collaborations.

4. Model Type-Checking and Policy Analysis

Based on the formal model defined in the previous section, we use the Alloy Analyzer to automatically verify the consistency of specific SMC collaborations. As typically done in model-checking we define *assertions* representing desired properties and ask the analyzer to find counter-examples where the properties do not hold. In this section, we present examples of analysis that can be performed using our model, such as verifying the role assignments when establishing a collaboration between SMCS, or verifying inconsistent policy deployment across distributed SMCs, where obligation policies do not always have a corresponding authorisation policy. Section 6 will introduce additional analysis types that we envisage, including both application-independent properties or properties specific to a particular application domain.

Consider the configuration illustrated in Figure 4. The *DoctorSMC* has an obligation policy *Obl*, which has *Doctor* and *Patient* as the respective subject and target of the policy, is triggered by the event *highHR*, to perform the *startECG* action. Note that *Doctor* and *Patient* are roles required by the *DoctorSMC*. On the other hand, the *PatientSMC* has an authorisation policy *Aut*, which defines *Doctor* and *Patient* as the subject and target of the policy respectively. This is a positive authorisation (labelled “*modality: Positive*”) that permits the execution of the *startECG* action. Here *Doctor* and *Patient* are roles required by the *PatientSMC*.

In terms of role assignments, the *DoctorSMC* provides *Interface2*, which is assigned to the local role *Doctor*. This is a local assignment, where the role *Doctor* is also the subject role of the obligation policy enforced by this SMC. In this example, the same interface is exported to be seen by the patient and assigned to the *Doctor* role in the remote *PatientSMC* (where *Doctor* is the subject role of the authorisation policy enforced by that SMC). Similarly, the *PatientSMC* provides *Interface0*, which is assigned to the local role *Patient* and then also exported and assigned to

the *Patient* role required by the remote *DoctorSMC* (where *Patient* is the target role of the obligation policy enforced by that SMC). Both policies are active (the label “*active*” is set in each policy).

Type-checking is required for policy activation, as the SMCs (i.e. their interfaces) assigned to a role must provide the functionality expected by the policies written in terms of that role. Note that in Figure 4, *Interface2* (assigned to the subject of the obligation) provides the event *highHR* (required to trigger the obligation policy), and *Interface0* (assigned to the target of the authorisation) provides action *startECG* (which is the action to be allowed execution by the authorisation policy). This is a valid policy configuration between SMCs because each obligation policy has a corresponding authorisation policy, which allows the action specified by the obligation to be executed. The assignments presented in Figure 4 ensure that the interfaces assigned to each role provide the right events and operations required by the policies, and that interfaces assigned to subject/target roles in an obligation are also assigned to subject/target roles in a matching positive authorisation policy. The type-checking restriction for the activation of obligation policies is illustrated in Figure 5. Figure 6 illustrates the predicate that determines whether all obligations have a matching positive authorisation policy.

We can also use this model to ask the analyzer to generate counter-examples of other invalid configurations. For example, we can easily obtain examples where obligation policies do not have the required events necessary for triggering them. We can further check whether in given circumstances, authorisations permit the execution of actions that would threaten the integrity of the SMC or whether failure or departure of devices leaves the SMC in a state where it is no longer able to fulfil its primary functions. This is particularly important when developing body area networks for health monitoring as the integrity of the sensor configuration influences the medical interpretation of the physiological parameters collected. More traditional types of analysis, such as detecting policy conflicts in the form of modality or application specific conflicts [11] can also be performed.

```

all conf: Configuration, smc1, smc2: conf.participants, obl: (conf.active & ConcreteObligation)
{
  (obl in (smc1.obligations + smc1.(conf.loading)))
  ⇒ some aut: (conf.active & ConcreteAuthorisation)
  {
    (aut in (smc2.authorisations + smc2.(conf.loading)))
    and (aut.modality in Positive)
    and (obl.action == aut.action)
    and ((obl.subject).~(conf.assignment) == (aut.subject).~(conf.assignment))
    and ((obl.target).~(conf.assignment) == (aut.target).~(conf.assignment))
  }
}

```

Figure 6. Determining whether obligations have a matching authorisation policy (consistent policy deployment): for any two SMCs, SMC_1 and SMC_2 , and for all active obligation policies, if SMC_1 has this policy, then there is an active positive authorisation policy in SMC_2 , which specifies the same action as in the obligation, and which has the same interfaces assigned to the subject and target roles in both policies.

5. Related Work

Several studies have looked at the (conflict) analysis of policies in various forms [12], [13], [14], [15], some of it based on Model-Checking techniques. Some of this work continues to date focussing on more complex policy languages and forms of analysis [16], [17]. In particular, [13], [17] use *event calculus* to represent temporal enforcement of policies, showing how the fulfillment or not of obligations affect the behaviour of the system and of other policies. In contrast to these studies our focus is not on the ability to detect policy conflicts, but to unambiguously specify the desired behaviour of interacting Self-Managed Cells and then verifying if these SMCs are able to enforce their policies (type checking events/operations to interfaces, all obligations are authorised, etc). This allows us to analyse their behaviour when interactions across several cells occur, which includes collaborations between SMCs but also extends naturally to compositions and federations of SMCs.

We have considered several alternatives for the formal specification of an SMC's behaviour including *pi-calculus* [18], *ambient calculus* [19], *channel ambient calculus* [20] and others (which model the computation operationally). In contrast, the declarative specification style used by *Alloy* [21] was simpler to use and the associated tool set offered rapid feedback through guided simulations on the possible system states. Also, its visualiser provided a very intuitive way of making sense of the solutions found, helping us in better designing and understanding the SMC interactions. This has enabled us to rapidly fine tune the specification and uncover omissions from the informal models developed earlier. We used this technique to specify predicates to represent the discovery and departure of SMCs, assignment and unassignment of SMCs to/from roles, policy loading and unloading, and policy activation and deactivation.

6. Concluding Remarks

We presented a formal specification for the verification of management policies in collaborations of Self-Managed Cells. Our formal model is based on *Alloy*, which allowed us to characterise declaratively the integrity constraints related to SMC management. Although *Alloy* relies on the *small scope hypothesis*, in practice many errors in the behaviour and interactions between SMCs can be uncovered in this manner.

We are using the formal model defined in this paper to automatically verify the consistency of specific SMC collaborations. Thus we can model policy-based SMC interactions and verify the correctness of these interactions before its actual implementation and deployment in physical devices. In this paper we presented a few examples of model verifications we are able to perform: we can type-check policy deployment across distributed SMCs (ensuring that the SMCs involved in the policy execution support the operations and events required by the policies), we can easily detect omissions from the configurations e.g. obligation policies do not have a corresponding authorisation policy, and we can analyse for application specific properties of an SMCs behaviour that may be violated due to dynamic changes such as the loading of new policies, the failures of components or the addition of new ones.

The formal model presented in this paper only caters for the elementary concepts such as SMCs, roles, interfaces and policies, and simple predicates for discovery of a new SMC, departure of an SMC, role assignment, role unassignment, policy loading and activation, etc. Our framework aims to provide the ability to extract collaboration patterns across multiple SMCs that can be re-used in frequently occurring situations. Work is in progress towards the formalisation of such complex patterns of interaction between SMCs [3] and the specification of safety properties for composing these patterns. However, their formalisation is also more complex

and could not be included here.

Our overall goal is to be able to dynamically form collaborations, compositions and federations of SMCs suitable to particular application scenarios e.g., such as care management for a set of patients, by instantiating combinations of pre-defined patterns. These application specific patterns can then be analysed and checked off-line prior to instantiation and deployment in order to minimise the occurrence of errors but also to increase performance by reducing the number of run-time checks that need to be performed.

Acknowledgment

Research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

- [1] E. Lupu, N. Dulay, M. Sloman, J. Sventek, S. Heeps, S. Strowes, K. Twidle, S.-L. Keoh, and A. Schaeffer-Filho, "AMUSE: autonomic management of ubiquitous systems for e-health," *J. Concurrency and Computation: Practice and Experience*, John Wiley, vol. 20(3), pp. 277–295, May 2008.
- [2] A. Schaeffer-Filho, E. Lupu, N. Dulay, S. L. Keoh, K. Twidle, M. Sloman, S. Heeps, S. Strowes, and J. Sventek, "Towards supporting interactions between self-managed cells," in *1st Int. Conf. on Self-Adaptive and Self-Organizing Systems (SASO)*. Boston, USA: IEEE Computer Society, July 2007, pp. 224–233.
- [3] A. Schaeffer-Filho, E. Lupu, and M. Sloman, "Realising management and composition of self-managed cells in pervasive healthcare," in *3rd Int. Conf. on Pervasive Computing Technologies for Healthcare*. London, UK: IEEE, April 2009.
- [4] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed., ser. Professional Computing Series. Addison-Wesley, 1995, 416 pages.
- [5] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proc. 5th European Software Engineering Conf.* London, UK: Springer-Verlag, 1995, pp. 137–153.
- [6] D. Garlan and M. Shaw, "An introduction to software architecture," in *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola and G. Tortora, Eds. Singapore: World Scientific Publishing Company, 1993, pp. 1–39.
- [7] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a taxonomy of software connectors," in *ICSE '00: Proc. 22nd Int. Conf on Software Engineering*. New York, NY, USA: ACM, 2000, pp. 178–187.
- [8] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Trans. on Software Engineering*, vol. 21, no. 4, pp. 314–335, 1995.
- [9] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.
- [10] —, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
- [11] E. Lupu and M. Sloman, "Conflicts in policy-based distributed systems management," *IEEE Trans. on Software Engineering*, vol. 25, no. 6, pp. 852–869, Nov. - Dec. 1999.
- [12] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *ICSE*, G.-C. Roman, W. G. Griswold, and B. Nuseibeh, Eds. ACM, 2005, pp. 196–205.
- [13] A. Bandara, E. C. Lupu, and A. Russo, "Using event calculus to formalise policy specification and analysis," in *IEEE 4th Int. Workshop on Policies for Distributed Systems and Networks (Policy 2003)*. Como, Italy: IEEE, June 2003.
- [14] M. Charalambides, P. Flegkas, G. Pavlou, J. Rubio-Loyola, A. K. Bandara, E. C. Lupu, A. Russo, M. Sloman, and N. Dulay, "Dynamic policy analysis and conflict resolution for diffserv quality of service management," in *NOMS*, J. L. Hellerstein and B. Stiller, Eds. IEEE, 2006, pp. 294–304.
- [15] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, G. Pavlou, and A. Lluch-Lafuente, "Using linear temporal model checking for goal-oriented policy refinement frameworks," in *POLICY*. IEEE Computer Society, 2005, pp. 181–190.
- [16] S. Jha, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough, "Towards formal verification of role-based access control policies," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 4, pp. 242–255, 2008.
- [17] R. Craven, J. Lobo, J. Ma, A. Russo, E. Lupu, A. Bandara, S. Calo, and M. Sloman, "Expressive policy analysis with enhanced system dynamicity," in *ACM Symp. on Information, Computer and Communications Security (ASIACCS 2009)*. Sydney, Australia: IEEE, March 2009.
- [18] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Inf. Comput.*, vol. 100, no. 1, pp. 1–40, 1992.
- [19] L. Cardelli and A. D. Gordon, "Mobile ambients," in *FoSSaCS '98: Proc. 1st Int. Conf. on Foundations of Software Science and Computation Structure*. London, UK: Springer-Verlag, 1998, pp. 140–155.
- [20] A. Phillips, "Specifying and implementing secure mobile applications in the channel ambient system," Ph.D. dissertation, Imperial College London, April 2006.
- [21] Tutorial for Alloy Analyzer 4.0, 2009, available at: <<http://alloy.mit.edu/alloy4/tutorial4/>>. Access in: February 2009.