F. Moyano, C. Fernandez-Gago, and J. Lopez, "A Model-driven Approach for Engineering Trust and Reputation into Software Services", *Journal of Network and Computer Applications*, vol. 69, pp. 134-151, 2016.
NICS Lab. Publications: https://www.nics.uma.es/publications

A Model-driven Approach for Engineering Trust and Reputation into Software Services

Francisco Moyano, Carmen Fernandez-Gago, Javier Lopez
Network, Information and Computer Security Lab
University of Malaga, 29071 Malaga, Spain
{moyano,mcgago,jlm}@lcc.uma.es

June 22, 2016

Abstract

The ever-increasing complex, dynamic and distributed nature of current systems demands model-driven techniques that allow working with abstractions and self-adaptive software in order to cope with unforeseeable changes. Models@run.time is a promising model-driven approach that supports the runtime adaptation of distributed, heterogeneous systems. Yet, frameworks that accommodate this paradigm have limited support to address security concerns, hindering their usage in real scenarios. We address this challenge by enhancing models@run.time with the notions of trust and reputation. Trust improves decision-making processes under risk and uncertainty and constitutes a distributed and flexible mechanism that does not entail heavyweight administration. This paper presents a trust and reputation framework that is integrated into a distributed component-model that implements the models@run.time paradigm, thus allowing the system to include trust in their reasoning process. The framework is illustrated in a chat application by implementing several state-of-the-art trust and reputation models. We show that the framework entails negligible computational overhead and that it requires a minimal amount of work for developers.

1 Introduction

Two important changes are coming to the Information and Communication Technology (ICT) world. On the one hand, the service-oriented vision enables on-the-fly improvements upon the functionality available to users. Applications are more dynamic and call for rapid adaptation strategies in order to meet new requirements and to respond to their changing environment. On the other hand, the boundaries between physical and virtual worlds are vanishing with the emergence of the Internet of Things (IoT), where sensors and actuators are embedded in daily life objects and are linked through networks capable of producing vast amount of data. The aforementioned reasons blur boundaries between design and runtime [7] as they prevent designers from envisioning all possible circumstances that might appear during the execution of an application.

The widespread adoption of these systems requires addressing three main concerns: complexity, dynamicity and security. The software engineering community is developing methods for addressing the first two concerns. In particular, model-driven engineering tames the complexity of systems by working with high-level models of them [28]. *Models@run.time* keeps abstract representations of running systems in order to reason about changes and drive dynamic reconfigurations [1], which tackle dynamicity.

One way to address security in these systems is by not taking for granted trust relationships among users, components, and system environments. These relationships must be explicitly declared, monitored and changed according to the system evolution. The contribution of this paper is the design and implementation of a trust and reputation development framework, together with its integration into a platform for self-adaptive, distributed component-based systems. The advantage of this integration is that reconfiguration decisions can be reasoned in terms of trust relationships and reputation information. As a result of such integration, developers can rely on a development framework that allows them to build highly dynamic, self-adaptive and trust-aware systems.

A remarkable issue about trust and reputation models today is that they often present high coupling with the application context, as they are designed as ad-hoc mechanisms that are plugged into existing applications, which in turn limits their reusability [4]. Therefore, one of the goals of our approach is allowing developers to implement different types of trust models. We achieve this by identifying high level concepts that form trust and reputation metamodels, and which abstract away from concrete instances. Then, developers can use these concepts as building blocks for their trust and reputation models.

The work presented here is an extension over our previous work [17]. The extensions include an implementation, considerations of reputation, and validation. This is done in two steps: first, we present a trust and reputation framework that allows developers implement a broad range of models; second, as we implement this framework on top of a self-adaptive platform, developers can use the output of the models in order to reconfigure the system at runtime.

The paper is structured as follows. Section 2 presents some works that are related to ours. An introduction to a models@run.time platform called Kevoree is given in Section 3. A brief discussion on trust and reputation concepts is presented in Section 4, whereas Section 5 describes the implementation of the framework. Section 6 presents our approach for allowing trust- and reputation-based reconfigurations of the system. A case study that illustrates the use of the framework in a chat application is described in Section 7. Section 8 yields experimental results as for the overhead and the amount of work that the development of such application requires, as well as the limitations of the framework and technical challenges that we faced. Finally, Section 9 concludes the paper by presenting some research challenges and lines for future work.

2 Related Work

There is an increasing interest in considering notions of trust in self-adapting systems in order to leverage reconfiguration decisions, especially in the areas of multi-agent,

component-based and service-oriented systems. In some cases, trust is considered in its hard variant, where trust is seen as an aggregation of quality of service (QoS) and security properties. In other works, the soft variant of trust is used, which means that social aspects such as reputation or preferences are taken into account [23].

As mentioned earlier, trust is seen as a powerful tool to leverage decision-making even with partial information. This fact is especially remarked in STRATUS [26], a set of technologies that aim at predicting and responding to complex cyber attacks. When it detects an attack, the platform switches to back-up components and finds alternative pathways of communication. The trust model that supports this platform [25] is based on conditional trust, that is, trust in certain capabilities of a system. The authors argue that experience-based trust is not useful because configurations in cyber attacks change frequently, laying statistical analysis useless. They propose ways to make the most out of the little information available, and introduce concepts like contagion that allows formalizing trust in a host based on the distance from an infected host. Even when the underlying idea is the same, they define a model, whereas in our work the model is only defined by the developer. Also, our framework is more general purpose and includes the option of implementing reputation models, whereas STRATUS is more specialized in cyber defence and does not tackle reputation.

A classical scenario of application of trust is multi-agent systems [22], where Vu et al. [29] propose trust-based mechanisms as a way to self-organize the agents in case of deceitful information. In particular, the trust value of an agent towards another one is an aggregate of direct experiences and testimonies. The use of artificial intelligence, and concretely, machine learning together with trust in order to adapt the behaviour of agents is proposed in the work by Klejnowski, Bernard, Hähner and Müller-Schloer [13]. They propose an architecture where there is an *observer* component that gathers information about the agent and presents it to the *controller* in two views: a long-term and a short-term one. The *controller* finds a suitable behaviour according to this information. Given that new unexpected situations might arise, agents must be able to try out new strategies and learn which ones provided the best results.

These works differ from ours in several aspects. They are framed within multi-agent systems, which means that each agent takes their own reconfiguration decisions. In our case, the system as a whole acts as a controller of its evolution. We do not consider machine learning techniques and we tackle both, trust and reputation. The most important difference is that whereas these works propose their own models, our framework is model-agnostic, meaning that it delegates to developers the responsibility of implementing trust or reputation models.

Given the highly open and distributed nature of service-oriented environments, the traditional use of trust is for either protecting providers from potentially malicious clients or for shielding clients against potentially malicious providers (e.g. providers that publish a higher Quality of Service (QoS) than offered). As an example of the first situation, Conner et al. [2] present a feedback-based reputation framework to help service providers to determine trust in incoming requests from clients. As an example of the second approach, Crapanzano et al. [3] propose a hierarchical architecture for SOA where there is a so-called super node overlay that acts as a trusting authority when a service consumer looks for a service provider.

In both, component- and service-oriented systems, an important research area is

determining the level of trust, or the trustworthiness, of the system as a whole, or of individual subsystems (i.e. services or components). In case that the trust value is too low, a reconfiguration takes place in order to try to improve it. In this direction, Haouas and Bourcier [8] present a runtime architecture that allows a service-oriented system to meet a dependability objective set up by an administrator. System dependability is computed by aggregating ratings provided by service consumers regarding QoS attributes. Then, a reconfiguration manager may look up other available services to meet the dependability objective. Dependability of the system is computed by the aggregation of each service dependability. In turn, each service dependability is computed by aggregating a weighted average of ratings provided by service consumers regarding QoS attributes (e.g. response time) of service providers. The reconfiguration manager is in charge of querying the service broker to find the available services that can meet the dependability objective.

Following a similar line of work in component-based systems, Yan and Prehofer [31] discuss an adaptive trust management system where several quality attributes can be used to rate the trustee's trustworthiness, such as availability, reliability, integrity or confidentiality. Assessing these attributes requires defining metrics and placing monitors to measure their parameters. Finally, trust is assessed at runtime based on the trustor's criteria and is automatically maintained by changing among trust control modes.

These previous works are highly focused on Quality of Service (QoS)-based trust, where trust is an aggregation of dependability and security attributes. Subjective factors affecting trust and reputation concepts, with which we deal, are out of discussion. The social notion of trust is used by Psaier et al. [21] in their self-adaptation framework. In particular, their trust model uses the concept of trust mirroring and trust teleportation. The former implies that actors (i.e. services) with similar interests and skills tend to trust each other more than unknown actors, whether the latter denotes that the level of trust in a member of a group is transferable to other members of the same group. The adaptations consist of reconfiguring the network by opening channels to provide new interactions and by closing channels to hinder misbehaving nodes to further degrade the system function. The trust model is used to help choose among a set of new candidate nodes with which to communicate.

Another way to measure the (mis)behaviour of components is by comparing its interactions with the models in their contracts. In this direction, Herrmann and Krumm [9] propose security wrappers that monitor the activity of the components. Depending on the deviation of the components' behaviour with respect to their contract, a positive or negative report is issued and sent to the trust information system, which calculates a trust value for the component. In turn, this trust value is used to determine the intensity of the monitoring activity by the wrappers. This scheme was enhanced by Herrmann [10] in order to take the reputation of components' users into account so as to prevent deliberate false feedbacks. In this regard, a common problem in any setting where different entities rate each other is discerning fair from unfair ratings. Phoomvuthisarn, Liu and Zhu [20] propose a reputation mechanism for SOAs environments that allows services to retrieve other services' reputation through auctions that ensure incentives for truthful reporting.

In our work, we do not use contracts and we are not concerned about fairness of

ratings, as this is something inherent to the model implemented, and thus the responsibility lies in the developer.

Other works focus on the self-adaptation of trust models to match and reflect the status of the system [14], and on considering the trust in the self-adaptation process itself [6]. The scope of our work is however different than theirs, because we are interested in adapting the system as a response of the trust information (and not vice-versa) and we are not concerned about trust in the self-adaptation process itself.

Kiefhaber et al. [12] present the Trust-Enabling Middleware, which provides applications running on top of it with methods to save, interpret and query trust related information. The middleware provides self-configuration and self-optimization and its goal is balancing the workload of nodes by relocating services. The middleware also uses built-in functions to measure the reliability of nodes by considering packets losses. Although this work shares many ideas with ours, there are two fundamental differences. First, it focuses on load balancing and the distribution of services among nodes. Second, it does not use the models@run.time paradigm, therefore reasoning about the current executing system is not straightforward.

As a conclusion of our literature review, the use of trust for assisting decision-making processes in self-adaptive systems is a topic of growing interest. However, we have found no framework that allows fast implementation of existing and custom trust and reputation models, and at the same time, an easy specification of self-adaptation actions based on trust and reputation information. Another common problem is that trust and reputation are often mixed up, when they are actually different although interrelated concepts. Our intent with this paper is to bridge these gaps.

3 Kevoree: A Models@run.time Development Platform

Traditionally, the Model-Driven Software Development area has primarily focused on using models at design, implementation and deployment phases of the Software Development Life Cycle (SDLC). However, as systems become more adaptable, reconfigurable and self-managing, they are also more prone to failures, which demands putting in place appropriate mechanisms for continuous design and runtime validation and monitoring. Models@run.time [1] aims to tame the complexity of dynamic adaptations by keeping an abstract model of the running system, pushing the idea of reflection one step further. The abstract model is synchronized with the actual system and every change performed on the model is automatically accommodated by the system.

Kevoree[1] is an open-source dynamic component model that relies on models at runtime to properly support the design and dynamic adaptation of distributed systems [5]. Six concepts constitute the basis of the Kevoree component metamodel. A *node* is an abstraction of a device on which software *components* can be deployed, whereas a *group* defines a set of nodes that share the same representation of the reflecting architectural model. A *port* represents an operation that a component provides or requires. A *binding* represents the communication between a port and a *channel*, which allows the communication among components. The core library of Kevoree implements these

[1] http://kevoree.org

concepts for several platforms such as Java, Android or Arduino. Fig. 1 depicts a snapshot of the aforementioned concepts in the Kevoree Editor, which allows building systems in a visual environment.

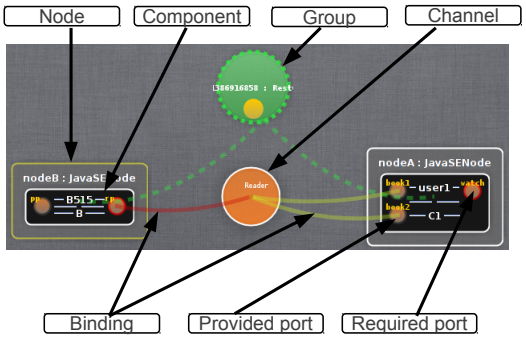

Figure 1: Kevoree Architectural Elements

Kevoree adopts the models@run.time paradigm and it boils down the reconfiguration process to moving from one configuration, represented by the current model, to another configuration represented by a target model. First, the target model is checked and validated to ensure a well-formed system configuration. Then, the target model is compared with the current model and this comparison generates an adaptation model that contains a set of abstract primitives that allow the transition from the current model to the target model. Finally, the adaptation engine instantiates the primitives to the current platform (e.g. Java) and executes them. If an action fails, the adaptation engine rollbacks the configuration to ensure consistency between the models@run.time layer and the running system.

Building an application with Kevoree entails two steps. First, developers create business components through the framework provided by the Kevoree platform. Second, components are deployed on nodes and wired together through bindings and channels. Next sections explain these steps in further detail.

3.1 Kevoree Development Framework

The framework is based on annotations. Components that run on the Kevoree runtime are created by annotating them with *ComponentType*[2].

Components can have *parameters*, which are attributes that are mapped to the reflection layer and can be changed at runtime via *Kevscript* or the visual editor. Additionally, components provide and require services through their ports.

Listing 1 defines a *Console* component with one required port, one provided port and one parameter. The parameter *showInTab* determines the appearance of the console frame and can be changed easily at any time both from the editor and with *Kevscript*, a script language provided by Kevoree. The required port *textEntered* allows sending

[2]In the same way, there are annotations to create new channels (*ChannelType*) and nodes (*NoeType*). For the purpose of this paper, however, we only need to create new components.

text to other consoles, whereas the provided port *showText* allows receiving text from other consoles and show it to the user.

Listing 1: Defining a new component with two ports

```
@ComponentType
public class Console {

    @Param(defaultValue = "true")
    protected Boolean showInTab = true;

    @Output
    protected Port textEntered;

    @Input
    public void showText(Object text)
    {
        //Show received text
    }
}
```

The services offered by the Kevoree runtime can be accessed by components through services injected at runtime. Requesting these services entail adding an attribute of the correspondent service type, and annotate such attribute with *@KevoreeInject*. For example, by using the *ModelService* type, developers gain access to the system model and can query it programmatically.

First, a list of the existing nodes in the model is retrieved, and for each of these nodes, we check its name with the searched name. If they are equal, all the components running on the node are retrieved. For each component, if the name of the component type matches the searched one, the instance name of the component is added to the result list, which is finally returned.

3.2 Deployment in Kevoree

Once business components are developed, they can be deployed in nodes and connected through ports. This deployment phase can be realised through the Kevoree editor or by *Kevscript*.

The editor provides a set of basic, built-in libraries (e.g. nodes, basic components and channels) and allows loading custom libraries (i.e. custom business components, customized nodes, channels, etc). It provides drag and drop functionality and a visual representation of the system architecture. The models can be converted to *Kevscript* instructions, being possible to save the model as a *.kevs* file containing these instructions.

As the complexity of the system increases, the editor may end up overloaded with too much information. In these cases, it is possible to deploy the system by manually specifying *Kevscript* instructions. Kevscript is a scripting language that allows the communication with the Kevoree reflection layer. It supports the addition and removal of nodes, components and channels at runtime.

The Kevoree platform does not support reasoning about security concerns, therefore any architectural element such as a node or a software component can join the system without further checks. Also, there is no cross-cutting criteria to guide the runtime changes. Our goal is to provide components with trust and reputation capabilities, which in turn can guide some reconfiguration decisions.

4 Trust and Reputation Concepts

This section provides an overview of the most relevant trust and reputation concepts. For greater insight on these concepts, we refer the reader to [18, 19].

The definition of trust is not unique as it often depends on the application context. In the scope of this work, we define trust as the personal, unique and temporal expectation that a trustor places on a trustee regarding the outcome of an interaction between them. This interaction usually comes in terms of a task that the trustee must perform and that can negatively influence the trustor. The expectation is personal and unique because it is subjective, and it is temporal because it may change over time.

The implications of trust are embodied in the so-called trust models, which are abstractions of the dynamics of trust and define the way to specify and evaluate trust relationships among entities for calculating trust. Another way of defining trust models is as the technical approach to represent trust for the purpose of digital processing [32].

The heterogeneity of trust models is due to several factors, including the trust definition on which they are built or their application domain. Trust models can be classified into two broad classes, which cover the two main branches that gave rise to the adoption of trust in the computational world.

- Decision Models. Trust management has its origins in these models [16]. They aim to make more flexible access control decisions, simplifying the two-step authentication and authorization process into a one-step trust decision. *Policy models* and *negotiation models* fall into this category. They build on the notions of policies and credentials, restricting the access to resources by means of policies that specify which credentials are required to access these resources.

- Evaluation Models. These models have their origin in the work by Marsh [15]. Their intent is to evaluate the reliability (or other similar attributes) of an entity by considering factors that have an influence on trust relationships. An important sub-class of the former are *propagation models*, in which existing trust relationships are exploited to generate new trust relationships. Another important sub-class are *reputation models*, where a reputation scored is derived from the aggregation of other entities opinions.

The framework presented in this work is built upon the concepts behind evaluation models. This means that the framework is tailored to support the implementation of evaluation models, rather than decision models. We present next the most important concepts[3] related to evaluation models, as well as some discussion on reputation concepts.

[3]Our previous work [18] gives details on the sources that we used for this analysis.

4.1 Evaluation Models Concepts

Trust is computed by a trust model that must have, at least, two entities which have to interact in some way. Entities play roles and in the most general case, these roles are trustor, the entity which places trust, and trustee, the entity on which trust is placed. However, depending on the context and complexity of the model, other roles are possible. For example, entities can be witnesses if they inform about their opinions of other entities based on observations or their own experience. Entities can also be factor producers, meaning that these entities generate factors that are considered by the trust model in order to yield trust values. Some specializations of trustors and trustees include requesters of services or resources, providers of services or resources, or trusted third parties that issue credentials or gather feedbacks to compute a centralized reputation score.

Once there is a trustor and a trustee, we say that a trust relationship has been established. Trust relationships are tagged with a trust value that indicates to what extent the trustor trusts the trustee. This value has a dimension, which indicates whether it is a single value or a tuple of values. Trust values are assigned to relationships by a trust assessment process. The concept of trust assessment, and all the concepts related to it, are the most important ones in evaluation models, as they may become the model signature, what makes a model different from others. In order to carry out the trust assessment process, trust metrics are used. Trust metrics use factors and combine them in order to yield a final score for the measured attributes. We can see in Section 7 different ways of trust metrics in the different models that are going to be implemented as examples of our framework. These metrics are specified in these cases mainly as mathematical functions. Factors are variables that influence a trust relationship and may include trustee's and trustor's subjective properties such as honesty, confidence, feelings, willingness or belief; and trustee's and trustor's objective properties like observed behaviour, security, ability, a given set of standards or reputation. Trust metrics use computation engines, which determine the way factors are combined, and which range from simple summations to more sophisticated ones like belief, Bayesian, fuzzy or flow engines. Sources of information that may feed the engines include direct experience (either direct interaction or direct observation), sociological and psychological factors, and third party referrals. Regardless of which information sources are used to compute trust values, the model might consider how certain or reliable this information is (e.g. credibility of witnesses), and might also consider the concept of time (e.g. how fresh the trust information is). Fig. 2, which is taken from [18], depicts the relationships among the discussed concepts and some other concepts that go beyond the scope of this work.

4.2 Reputation Concepts

Reputation models use public trust information from other entities to yield a reputation score. Whereas the boundaries between trust and reputation are often blurry in the literature, it is agreed that reputation is a factor that may influence trust decisions [11]. We largely base our reputation terminology on web reputation systems [4].

Reputation models can be centralized, when there is an entity in charge of collecting

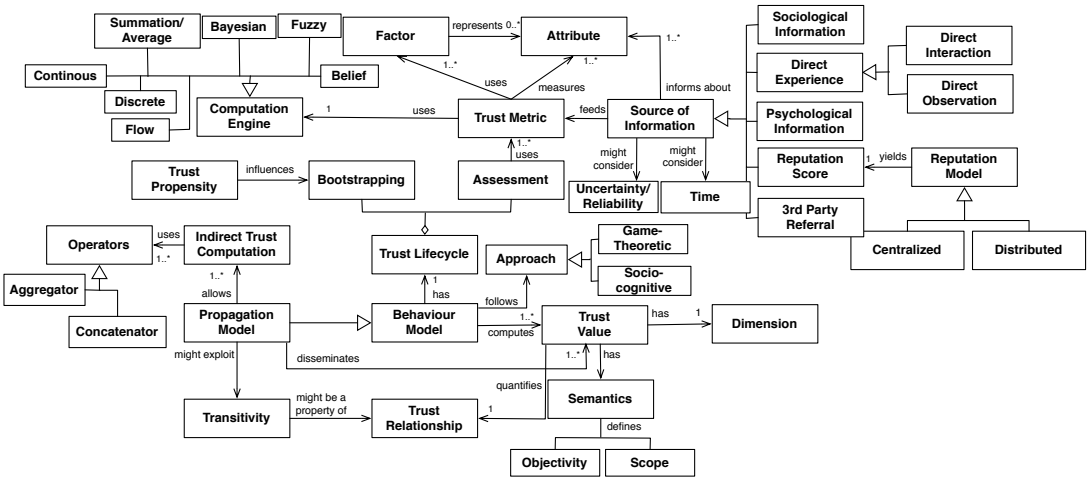

Figure 2: Concepts for Evaluation Models

and distributing reputation information; or distributed, when there is no such a role and each entity has to maintain a record of trust values for any other entities, and send out this information to the rest of entities. A reputation statement is a tuple of a source entity, a claim and a target entity. A source entity is any entity in the system capable of making claims about another entity of the same system, which is called a target entity. A claim is a statement made by a source entity about a target entity. Reputation models use reputation engines that take reputation statements about a given target as inputs and yield a reputation score for that target.

5 Integrating Trust and Reputation in Models@Run.time

The main contribution of this paper is a framework that integrates the notions of trust and reputation into the Kevoree component model described in Section 3, and is an extension over our previous work [17]. This framework consists of several classes that developers can customize by inheritance and by overriding or implementing some of their methods [4]. These classes use the Kevoree framework classes as the fundamental building blocks in order to make the integration with Kevoree possible. Developers using our framework can build trust and reputation models right into the models@run.time platform provided by Kevoree, which in turn enables that trust and reputation information can be used in order to make reconfiguration decisions at runtime. Developers can use this framework in order to build trust and reputation models for self-adaptive systems. Trust and reputation information that these models generate can be used to make reconfiguration decisions.

[4]The interested reader can check more information about the Javadoc or source code in `https://www.nics.uma.es/development/trust-and-reputation-framework-modelsruntime`

The framework consists of an API for developers with some base components that can be extended, some methods that can be overridden, and configuration files. The rest of this section describes the most important aspects of the framework implementation and its integration in the Kevoree component model.

5.1 Trust and Reputation Metamodels

We use the Eclipse Modelling Framework (EMF)[5] to create metamodels for trust and reputation. These metamodels gather a set of concepts and relationships among these concepts that abstract away the particularities of different trust models, in such a way that different metamodels instantiations yield different models. Fig. 3 and 4 show the trust and reputation metamodels, respectively.

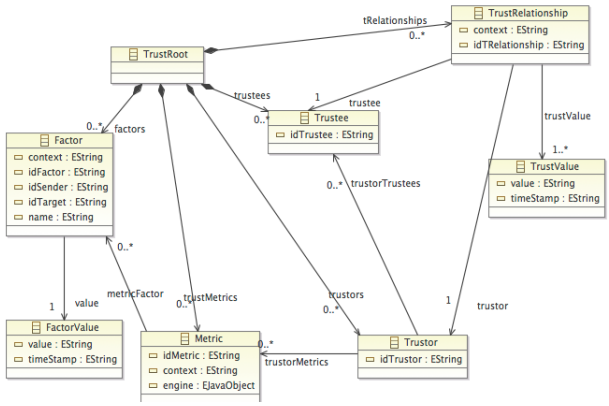

Figure 3: Trust Metamodel

The trust metamodel includes the concept of *TrustRelationship*, which is a tuple of a *Trustor*, *Trustee* and *TrustValue*. *Trustor*s use *Metric*s to evaluate their trust in *Trustee*s. *Metric*s use a set of *Factor*s, which in turn have a *FactorValue*. Different trust models are created by instantiating the entities that play the trustor and trustee roles, the factors that are considered and the way these factors are combined in a metric.

The core concept of a reputation metamodel is a *ReputationStatement*, which is a tuple containing a *Source* entity, a *Target* entity and a *Claim*, which has a *ClaimValue*. A *ReputationMetric* is used in order to aggregate *Claim*s. Reputation models are created by instantiating the entities that play the source and target roles, the way claims are generated and their type, and the way the metric combines the claims.

Note that these metamodels gather many of the concepts explained in Section 4. In both metamodels, other important concepts from the conceptual framework are included as attributes, like *Context* and *Time*. Other concepts from the conceptual framework that are not presented explicitly in the metamodel are included implicitly in the implementation. For example, factors can be objective and subjective, but the differ-

[5]http://www.eclipse.org/modeling/emf/

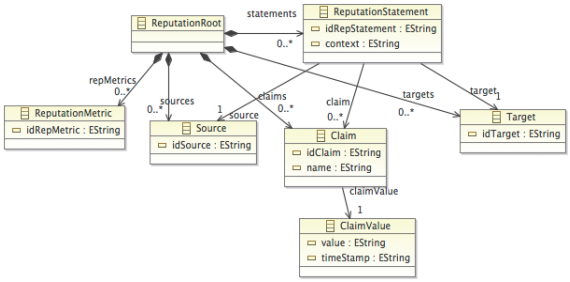

Figure 4: Reputation Metamodel

ence is only made at the implementation level with methods available to entities, such as `addSubjectiveFactor`. *Engine*s are concrete implementations of *Metric*s.

Another example concerns centralized and distributed reputation models. As we see in the next section, centralized reputation models include entities that must send their claims to a component that stores them and which compute reputation, whereas distributed reputation models comprise entities that store their own claims and which compute reputation themselves. In summary, metamodels provide a basic skeleton of relevant concepts, which are enriched during implementation to accommodate more concepts discussed in the conceptual framework

From these metamodels, the EMF generates code that constitutes an API to manage these metamodels. This code does not need to be visible to developers, who can be oblivious about how trust models are instantiated and managed internally by the framework. We use this code as an internal API that acts as an interface between the trust and reputation components and the underlying trust or reputation model.

The following sections describe the trust and reputation components, respectively, that constitute the framework[6].

5.2 Trust Framework

This section describes how the trust part of the framework is implemented. As mentioned earlier, this implementation is hidden from developers, as they do not need to know the implementation details in order to use the framework.

One of the main components in the trust framework is *TrustEntity*, which describes an entity capable of participating in a trust relationship. That is, each business component that we want to include in the trust dynamics must inherit from this component. Listing 9 shows an excerpt of the implementation.

We use Java generics so as to allow developers to set the types for the trust values and the factor values, respectively. We define several parameters for this component. The *role* parameter states whether the entity plays a trustor role, a trustee role, or both roles. The entity can also specify a *trust context* where its relationships are framed. Entities can belong to *group*s and their trust relationships are to be initialized according

[6]In the next sections, we will refer to some listings that are in Appendix A for readability's sake.

to the value of *bootstrappingTrustValue* parameter. The last parameter denotes the name of a file containing *subjective factors* information during initialization.

A trust entity requires services in order to update a trust relationship through the port *requestTrustUpdate*, to initialize its trust relationships through *initTrustRelationships*, and to add factors through *addFactor*. In the *start()* method, which will be called by the Kevoree framework at start-up, a unique identifier for the component is generated by the context service of Kevoree, which provides some basic context information such as the name of the instance and the node where the instance is running. Then, a request to initialize trust relationships is sent to a *TrustManager* component, and finally subjective factors are stored in the model.

Subjective factors are initialized by means of a file that the developer can configure for each trust entity. The format of the file is:

FactorName FactorValue <TargetEntity>

The last parameter is optional and denotes the entity to which the subjective factor applies. For instance, if an entity A thinks that another entity B is competent, the file with A's subjective factors would include:

PerceivedCompetence High B

The most important methods offered by the *TrustEntity* component are *changeSubjectiveFactor()*, *requestTrustUpdate()*, and *trustRelationshipUpdated()*. The former allows trust entities to increment or decrement an existing subjective factor or to create a new one. The second one requests an update of a trust relationships with a trustee, and the latter is called by the framework when the update is done. Clients can invoke the first two methods and can override the last one in order to make business-level decisions based on trust values. By default, when the client calls *requestTrustUpdate()*, reconfiguration of the system might occur. An overloaded version of the method allows explicitly inhibiting a reconfiguration after the update by setting a parameter to *false*. The method *getLastTrustValue()* provides access to recently computed values.

TrustEntity components interact with a *TrustModel* component, which manages the trust metamodel and is in charge of computing trust values. Listing 10 shows how the initialization of trust entities relationships is performed. The key of this method is the call to the static method *getTrusteesInstanceName()*, provided by the *GetHelper* class, which is a utility class that allows querying and retrieving information from the reflection layer. We consider that an entity is trustee with respect to another entity if it plays that role and they share the same context. The last line of code calls an internal method in order to add the trust relationship to the metamodel, for which the EMF API (see Section 5.1) is used, as depicted in Listing 11.

Clients of *TrustModel* can invoke several methods in order to retrieve factor information, and can override two methods, *compute()* and *computeThreshold()*. This is illustrated in Listing 12. Retrieving factor values is essential in order to implement the trust engines, which need these values to compute trust. Trust engines are implemented by overriding the *compute()* method, and optionally, the *computeThreshold()* method.

The last important component of the the trust framework is the *FactorProducer*[7]. This type of entity adds objective factors about other entities by using low-level platform services that provide information about the components and their communications. This is a key component to QoS-based trust models, as it allows the model to easily take into account information about the response times, number of failures, uptime percentage of services, and so on.

Clients of this component must set the instance identifier of the target entity and override the method *doEvaluation()*. This component can work in two ways: by assigning a value at initialization time, and by monitoring the target at a regular interval that developers can also specify in another parameter. The value returned by the method is automatically included as a factor in the method, and engines can retrieve the factor during the computation.

The next section revises some implementation details of the reputation framework.

5.3 Reputation Framework

The reputation framework allows the implementation of centralized and distributed reputation models by means of *CentralReputableEntity* and *DistReputableEntity* components, respectively. The former requires the communication with a *ReputationManager* component in order to send to it the claims and retrieve reputation information, whereas the latter requires a *ReputationEngine* that will be in charge of computing reputation for the component.

Part of the *CentralReputableEntity* implementation is presented in Listing 13. Again, reputation takes place in a *trust context*, and entities may belong to a *group* and need a unique identifier *uid*. These entities also require a port through which to send their claims and request reputation information. Two important methods offered to clients are *makeClaim()* and *requestReputation()*, which allow sending claims to and retrieving reputation information from the *ReputationManager*. Another important method that client code can override is *reputationReceived()*, as depicted in Listing 13. This method is called by the framework when a new reputation value of an entity is computed. As in the case of *TrustModel*, when a reputation update is requested, the default behaviour is reconfiguring the system in case it is required, although the client can explicitly disable the reconfiguration by setting the corresponding parameter to *false*. *CentralReputableEntity* also caches the last computed reputation values in order to provide faster access, through the method *getLastReputation()*, to the reputation of an entity with which it interacted in the past.

The other important component of a centralized reputation model is the *ReputationManager*, which interacts with the reputation metamodel and offers methods to retrieve claims information. It also provides the method *compute()*, which clients must implement for their reputation engines. An excerpt of the implementation is depicted in Listing 14. As an example, the method *getClaimValues()* retrieve the values of all the claims that are issued in a specific *context*, with a specific *name* and for a concrete *target* entity. Clients can inherit from this class and override the method *compute()*.

[7]This component is explained for completeness. However, in the scope of this work we are interested in the social notions of trust and reputation, therefore the chosen models in Section 7 do not use this component.

The other type of reputation model, namely distributed reputation models, are built around two components: *DistReputableEntity* and *ReputationEngine*. The former represents an entity capable of issuing claims (such as *CentralReputableEntity*), but which is responsible to store its own claims and to send them to other entities that may request them. The latter is a reputation engine that belongs to a distributed entity. A reputation engine is bound to an entity at start-up, as illustrated in Listing 15. As in the case of the *CentralReputableEntity*, this component provides several methods to issue claims and to request reputation updates. Clients can also override methods to react when a new reputation value arrives.

Clients using *ReputationEngine* must implement the method *compute()* and can gain access to the claims in the system by methods like the one shown in Listing 16, which retrieves all the claims made by an entity about a particular subject, represented by the name of the claim.

Next section discusses how the framework provides trust-based self-adaptation.

6 Trust-based Self-Adaptation

The interesting advantage of implementing the framework on top of a self-adaptive platform is that developers can use trust and reputation information to change the system at runtime. Regardless of the implemented model, the output of the model (i.e. the trust or reputation value) can be used to make reconfiguration decisions.

6.1 Policy-based Reconfiguration

The framework supports the reconfiguration process by means of policies in the form of simple rules. Rules for reputation-based reconfiguration are as follows:

CType BooleanCondition Action <Args>

where *CType* is the type of the component for which the reputation is to be used in the *BooleanComparison*. If this comparison is true, then *Action* is executed with the required arguments (*Args*). Trust-based reconfiguration rules are similar:

CType CType BooleanCondition|threshold Action <Args>

where the first *CType* is the type of the trustor and the second one is the type of the trustee. Either the trust value is compared according to the *BooleanComparison* or the model threshold is used to determine whether the *Action* should be executed with the required *Args*.

The actions currently implemented in the framework are *remove* and *substitute*. The former does not require arguments and tells the runtime system to remove the component if the boolean condition is met. The latter requires an argument, which is another component type, and tells the system to substitute the current component by another component of the new type if the condition is fulfilled.

Let us illustrate with a couple of examples. Consider the following reputation-based reconfiguration policy file:

Console <1 substitute FilteredConsole
FilteredConsole <0 remove

This policy is specifying the following: "if the reputation of any instance of type *Console* is less than 1, then substitute it by another component instance of type *FilteredConsole*. Likewise, if the reputation of any instance of type *FilteredConsole* is less than 0, then remove it".

Consider now the following trust-based reconfiguration policy file:

Console Console threshold substitute Console
Console Console <0 remove

The policy is specifying the following: "if a trustor of type *Console* does not trust a trustee of type *Console* over a threshold (defined by the model), then substitute the trustee by a new component of type *Console*. Likewise, if the trust that a trustor of type *Console* places in a trustee of type *Console* is less than 0, then remove the trustee".

In addition to the new component type, the *substitute* action can have an undefined number of parameters that represent attributes of the new instances and their values. If no parameters are found, it is assumed that new instances should replicate the same values of the attributes of the instances removed.

As an example, consider the following:

Console <1 substitute FilteredConsole group A
FilteredConsole >5 substitute Console

This policy is specifying the following: "if the reputation of any instance of type *Console* is less than 1, then substitute it by another component instance of type *FilteredConsole* and set its parameter *group* to the value *A*. Likewise, if the reputation of any instance of type *FilteredConsole* is more than 5, then substitute it by a component instance of type *Console* and set all the parameters that have the same name to the same values that the previous instance had".

6.2 Implementation

In this section, we explain some details of the implementation. The main class is *ScriptEngine*, which encapsulates the actions and generates the *Kevscript* instructions. The reputation framework provides the class *ReputationRulesEngine*, which process the policy file and calls the script engine to generate the adaptation script. Likewise, the trust framework uses the class *TrustRulesEngine* for the same purpose. Listing 17 shows the initialization of *DistReputableEntity*. Note that instances of *ScriptEngine* and *ReputationRulesEngine* are created, and that the former is passed as an argument to the latter, together with the name of the policy file (RepRules.policy by default). The listing also shows an internal method of the framework which calls the *compute()* method of the reputation engine and which determines whether the user wants to reconfigure the system, in which case the method *executeRules()* of the *ReputationRulesEngine* class is called. In turn, this method reads the file and executes, via the instance of *ScriptEngine*, the rules for which the boolean condition is met.

As an example of how *ScriptEngine* executes Kevscript instructions, Listing 18 shows the implementation of the action *remove*. Once the name of the instance in the reflection layer is obtained (by trimming the name of the node where it is executing), the script is executed by means of the *ModelService* variable *model*, which allows submitting scripts as a *String* and checking the results in a callback.

Substituting a component entails more work, as a new component must be created and must be ensured that it will be able to continue its communication with the rest of components. Given that the code is much longer, we simply enumerate the steps required for this action[8]:

1. Obtain information (if necessary) about the attributes of the instance to be removed.
2. Obtain information about the bindings and channels of the component to be removed.
3. Remove component.
4. Create new instance name of the type specified in the policy file.
5. Add the component and link it to the channels via new bindings.
6. Add new attributes, which could be the same as the ones of the previous instance.

7 Application Example: A Trust-Aware Distributed Chat

In this section, we explain how we can implement several well-known trust and reputation models. First, we provide a brief description of each model followed by high-level steps required to implement them. We also show the most relevant code for the implementation of each model[9].

The chosen scenario is a distributed chat application, because it is a simple scenario that allows illustrating the use of trust and reputation models easily. The mechanics are similar for every model: a console receives a message from another console and inspects the contents of the message. Depending on these contents (e.g. if it detects a swear word), it provides a stimuli to the trust or reputation model. This stimuli may be claims or changes in factors, as it will be illustrated in each model.

N. B., that it is out of the scope of the paper to determine how the trust and reputation values are obtained for the models we are describing next. N.B. also some of the chosen models used both, trust and reputation values. Our framework does not allow to integrate a reputation model into a trust model that uses reputation, therefore we have to assume that the trust and reputation values are given to us.

[8]The interested reader can check more information about the Javadoc or source code in `https://www.nics.uma.es/development/trust-and-reputation-framework-modelsruntime`

[9]We omit some error checking and boiler plate code for the sake of better understandability.

7.1 eBay Model

In the eBay reputation model, after a transaction occurs, both the seller and the buyer can rate each other by a positive (-1), neutral (0) or negative feedback (1). The reputation for an individual is then calculated by summing up the distinct ratings for such individual [24]. The model is centralized, in such a way that all the feedbacks are sent to a central system that computes the reputation, and each user can query and see this reputation in the form of an html page.

This model is mapped to our example as follows. Once a console receives a message from another console, it looks for offensive words contained in the message, and which are previously configured by the user. If any offensive word is included, then a negative feedback about the sender is issued (-1); otherwise, a positive feedback is sent(1).

The implementation of this model in the framework requires the following coarse-grained steps:

- Consoles must inherit from *CentralReputableEntity*,
- Consoles invoke the method *makeClaim* upon receiving a message.
- Reputation engine inherits from *ReputationManager* and overrides the method *compute()*.

Further details are provided in Listing 2. A console component inheriting from a central reputable entity is created. The generic type is instantiated to *String* because that is the format in which we want to represent the reputation value. Then, upon receiving a message through the input port showText, it tokenizes the message to retrieve the identity of the sender and the text itself (provided by the console or by the middleware itself), and if any offensive word is found, it issues a negative claim with name *CleanWords*; otherwise, it issues a positive one. The message is temporarily stored in *lastMessageReceived* (if there are no offensive words) and an update of the reputation of the sender is requested, which will call the reputation engine. In turn, the reputation engine will automatically call the method *reputationReceived* with the identity of the entity and the new reputation value. In this method, we could perform additional checks to determine whether the console should print the message or not.

Listing 2: Console in the Ebay Reputation Model

```
@ComponentType
public class CentralReputationAwareConsole extends
    CentralReputableEntity<String> {

   //It stores the last message receive
   private String lastMessageReceived;

   @Input
    public void showText(Object text)
    {
        if( text != null ) {
            String msg = text.toString();
```

```
            StringTokenizer st = new StringTokenizer( msg, "."
                );
            String idTarget = st.nextToken();
            String message = st.nextToken();
            if( badWordsInMessage( message ) )
            {
               lastMessageReceived = "";
               makeClaim("CleanWords", "-1", idTarget);
            }
            else
            {
                lastMessageReceived = message;
                makeClaim( "CleanWords", "1", idTarget );
            }
            requestReputation( idTarget );
        }
    }

    @Override
    public void reputationReceived( String target, String
        newVal )
    {
      //We could check if the reputation is above a given
          threshold prior to showing the message
        thisConsole.appendIncomming( "->" + lastMessageReceived
            );
    }

}
```

The reputation model, depicted in Listing 3, implements the Ebay reputation engine. First, it retrieves all the claims named *CleanWords* about the target *idTarget*. If there are no claims about the target, then a default value is returned, otherwise, the reputation value is computed by summing up all the claim values.

Listing 3: Ebay Reputation Engine

```
@ComponentType
public final class EBayReputationModel extends
    ReputationManager<String> {

    @Override
    public String compute( String context, String idTarget,
        String idSource ) {

        List<String> claims = getClaimsValues( context, "
            CleanWords", idTarget );
        float res = 0.0f;
        if ( claims != null )
        {
         //By default reputation
```

```
            if( claims.size() == 0 )
            {
                return "1.0"; //Initial reputation
            }
            for ( String c : claims )
            {
                res += Float.valueOf( c );
            }
        }
        return String.valueOf( res );
    }
}
```

7.2 Marsh's Trust Model

Marsh was one of the first authors that formalised trust in a computational setting [15]. His model considers the following factors:

- Utility (U_x): this factor measures the utility that entities would obtain from a successful collaboration.
- Basic trust or trust disposition(T_x): this subjective factor indicates what is the attitude of an entity towards higher or lower values of trust.
- Importance (I_x): this subjective factor indicates how important a situation is for an entity.
- Perceived Competence: this subjective factor states how competent the trustor thinks that the trustee is for the task in play.
- Perceived Risk: this subjective factor denotes how risky the entity thinks the situation is.
- General trust ($T_x(y)$): this refers to the trust that the trustor places in the trustee as a consequence of the history of interactions.

The model uses the aforementioned factors to calculate the so-called *situational trust*, which according to the author is the most important when considering trust in cooperative situations. Situational trust is defined as:

$$T_x(y,\alpha) = U_x(\alpha) \times I_x(\alpha) \times \widehat{T_x(y)}$$

where x is the trustor, y is the trustee, and α is the situation[10]. Marsh also models what he calls a *cooperation threshold*:

$$CT_x(\alpha) = \frac{PerceivedRisk_x(\alpha)}{PerceivedCompetence_x(y,\alpha) + \widehat{T_x(y)}} \times I_x(\alpha)$$

[10]The situation for Marsh is what we call context.

The model states that an agent engages in a collaboration with another agent if the situational trust is greater than the cooperation threshold. In order to implement the model in the framework, the following steps must be performed:

- Consoles inherit from *TrustEntity*.
- Consoles add their subjective factors.
- Consoles change their factors in response to the received messages.
- Trust engines inherit from *TrustModel* and overrides the methods *compute()* and *computeThreshold()*.

In order to initialize their subjective factors, we create a simple text file with a list of *factor value* <*target*>, where *factor* represents the factor name, *value* denotes the value of the factor and *target*, which is an optional parameter, the name of the component instance to which the factor refers. An example of this file for one of the consoles in Marsh's is illustrated in Table 1. The name of this file is assigned in Kevscript or from the editor for each console. Once the trust relationships of the console have been initialized, the file is read and the factors are stored.

Table 1: Trust Factors for Marsh's Model

Factor	**Value**	**Target**
utility	1.0	-
trustDisposition	0.8	-
importance	0.4	-
perceivedRisk	0.2	-
perceivedCompetence	0.9	console234@node0

Listing 4 shows the console implementation. First, it inherits from *TrustEntity*, the generics of which are instantiated to String and Float, because this is the format we are representing the trust values and the trust factors, respectively. When a console receives a new message and after retrieving metadata (e.g. the sender, which is the trustee of the relationship), it stores the message and looks for offensive words in the text. If any offensive word is found, it invokes the method *changeSubjectiveFactor()*. The arguments tell that the factor *perceivedCompetence* that refers to the *trustee* entity should be decreased by -0.1, and in case this factor does not exist, it should be initialized to 0.5[11]. Finally, the console requests a trust update about its trustee.

When the trust update is completed by the trust engine, this will call the method *trustRelationshipUpdated()*, which indicates the trustee to which it refers, and a list with two potential values. The first value represents the actual new trust value, whereas the second value is the threshold value[12]. Depending on their relationship, the message is finally printed or not.

[11]The factor may not exist if a reconfiguration has taken place and a new component has been added for which there is not such factor.

[12]We say potentially because not all trust models include a trust threshold computation and therefore, in that case, it would be up to the developer to hard-code a reasonable threshold.

Listing 4: Console in Marsh Trust Model

```
@ComponentType
public class TrustAwareConsole extends TrustEntity<String,Float
    >

   //...

   @Input
    public void showText(Object text)
    {
        if( text != null ) {
            String msg = text.toString();
            StringTokenizer st = new StringTokenizer( msg, "."
                );
            String trustee = st.nextToken();
            String group = st.nextToken();
            String message = st.nextToken();
            lastMessageReceived.put( trustee, message );

            if( badWordsInMessage( badWords ))
            {
                changeSubjectiveFactor("perceivedCompetence",
                    -0.1f, 0.5f, trustee );
            }
         requestTrustUpdate( trustee );
        }
    }

    @Override
    protected void trustRelationshipUpdated(final String
        trustee, List<String> newVal) {

        float trustValue = Float.valueOf(newVal.get(0));
        float threshold = Float.valueOf(newVal.get(1));

        if( trustValue >= threshold )
        {
            thisConsole.appendIncomming( lastMessageReceived.
                get( trustee ) );
        }
    }
}
```

The trust engine is shown in Listing 5. First, it inherits from *TrustModel* and instantiates the generics to String and Float, which again are the formats of the trust values and trust factors. Here, the developer must implement the methods *compute()* and *incrementFactor()* and can implement the method *computeThreshold()*. The former computes a trust value from the trust factors as discussed earlier in the description of the model. The second method determines how an increment/decrement should be

performed depending on the concrete generics instantiation. The latter allows computing a threshold from the trust factors as discussed earlier in the description of the model.

Listing 5: Trust Engine for Marsh's Model

```
@ComponentType
public class MarshModel extends TrustModel<String, Float>
{

    @Override
    public String compute(String context, String idTrustee,
        String idTrustor)
    {
        float utility = Float.valueOf( getFactorValue( context,
            "utility", idTrustor ));
        float importance = Float.valueOf( getFactorValue(
            context, "importance", idTrustor));
        String generalTrustString = getFactorValue( context, "
            generalTrust", idTrustor, idTrustee );
        float generalTrust = Float.valueOf( getFactorValue(
            context, "generalTrust", idTrustor, idTrustee ));

        float situationalTrust = utility * importance *
            generalTrust;

        addFactor( context, "generalTrust", situationalTrust,
            idTrustor, idTrustee );
        return String.valueOf(situationalTrust);
    }

    @Override
    protected String computeThreshold(String context, String
        idTrustee, String idTrustor)
    {
        float perceivedRisk = Float.valueOf( getFactorValue(
            context, "perceivedRisk", idTrustor ));
        float perceivedCompetence = Float.valueOf(
            getFactorValue( context, "perceivedCompetence",
            idTrustor, idTrustee ));
         float generalTrust = Float.valueOf( getFactorValue(
             context, "generalTrust", idTrustor, idTrustee ));
        float importance = Float.valueOf( getFactorValue(
            context, "importance", idTrustor));

        float threshold = importance * perceivedRisk / (
            perceivedCompetence + generalTrust );

        return String.valueOf( threshold );
    }
```

```
    @Override
    protected String incrementFactor(String currentValue,
        String increment) {
        return String.valueOf( Float.valueOf( currentValue ) +
            Float.valueOf( increment ));
    }
}
```

7.3 PeerTrust

Xiong and Liu [30] propose PeerTrust, a distributed reputation model oriented towards Peer-to-Peer scenarios. This model, as in the case of most reputation models, builds a reputation score upon feedbacks that peers yield after their collaboration. In addition to the feedbacks, the model proposes using the following factors:

- Number of transactions that a peer has had with another peer.
- Credibility of the feedback; feedbacks from more trustworthy peers should weight more in the calculation.
- Transaction context, which refers to metadata about the context where the transaction or collaboration is taking place.
- Community context, which relates to incentives for providing feedbacks.

The general trust metric is the following:

$$T(u) = \alpha \cdot \sum_{i=1}^{I(u)} S(u,i) \cdot Cr(p(u,i)) \cdot TF(u,i) + \beta \cdot CF(u)$$

where $I(u)$ is the total number of transactions that peer u had with the rest of peers, $p(u,i)$ denotes the other participating peer in peer u's ith transaction, $S(u,i)$ is the satisfaction peer u receives from $p(u,i)$, $Cr(v)$ denotes the credibility of the feedback submitted by v, $TF(u,i)$ is the transaction context factor for u's ith transaction, and $CF(u)$ denotes the community context factor for peer u. α and β are weights for the collective evaluation and the community context factor.

In our example, we identify each transaction with a message sent and received by two communicating consoles. We lay the community context aside (i.e. $\beta = 0$) and focus entirely on the collective evaluation (i.e. $\alpha = 1$). The authors of the model provide hints about how to calculate the credibility and the context factor. In particular, the credibility can be calculated using the following formula:

$$Cr(u) = \frac{T(p(u,i))}{\sum_{i=1}^{I(u)} T(p(u,i))}$$

which uses the ratio between the current reputation of the peer that sent the satisfaction feedback and the total reputation of all peers that previously collaborated with u.

Regarding the context factor, the authors mention that a time-stamp of the transaction can be used in order to give more relevance to more recent transactions. One way to model this is by the following formula:

$$TF(u,i) = \frac{TS(u,i)}{CurrentTime}$$

where $TS(u,i)$ is the time when the ith transaction took place.

Once we have this, we can implement the model in the framework following these high-level steps:

- Consoles must inherit from *DistReputableEntity*,
- Consoles invoke the method *makeClaim* upon receiving a message.
- Reputation engine inherits from *ReputationEngine* and overrides the method *compute()*.
- Console is assigned the reputation engine created.

The console code is similar to the one shown in Listing 2. The difference is that in this case we are dealing with a distributed reputation model, therefore consoles must inherit from *DistKevReputableEntity* and each console is responsible to compute reputation values, instead of delegating this task to a reputation manager. Thus, in the start method of the console, we need to specify which reputation engine the console will use, as depicted in Listing 6.

Listing 6: Binding Reputation Engine and Console

```
@ComponentType
public class DistReputationAwareConsole extends
    DistKevReputableEntity<Float>

     @Start
    public void startConsole()
    {
        super.start( new PeerTrustModel() );

        //More console-specific initialization stuff
    }
}
```

The code for the reputation engine is illustrated in Listing 7, which implements the formula described earlier for PeerTrust. Note that the method *calculateTotalReputation()* is not part of the framework, but a way to modularize the *compute* method.

Listing 7: Binding Reputation Engine and Console

```
public class PeerTrustModel extends ReputationEngine<Float>
```

```
{
    @Override
     public Float compute(String context, String idTarget,
         String idSource)
     {
         List<ReputationStatementInfo> allStatementsAboutTarget
             = getClaimsAboutTarget( "CleanWords", idTarget );
         float totalReputation = calculateTotalReputation(
             idTarget );
         double currentTime = (double) Calendar.getInstance().
             getTime().getTime();
         float targetReputation = 0.0f;
         for ( ReputationStatementInfo rs :
             allStatementsAboutTarget )
         {
             String source = rs.getSource();
             float sourceReputation = Float.valueOf(
                 getLastReputation(source) );
             float credibility = sourceReputation /
                 totalReputation;
             targetReputation += Float.valueOf( rs.getClaim().
                 getValue() ) * credibility * Double.valueOf( rs.
                 getTimeStamp() ) / currentTime;
         }
         return targetReputation;
     }

     private float calculateTotalReputation( String idTarget )
     {
         List<ReputationStatementInfo> allStatementsAboutTarget
             = getClaimsAboutTarget( "CleanWords", idTarget );
         Set<String> consideredEntities = new HashSet();
         float totalRep = 0.0f;
         for ( ReputationStatementInfo rs :
             allStatementsAboutTarget )
         {
             String sourceEntity = rs.getSource();
             String sourceReputationString = getLastReputation(
                 sourceEntity );
             float sourceReputation = Float.valueOf(
                 getLastReputation( sourceEntity ) );
             consideredEntities.add( sourceEntity );
             totalRep += sourceReputation;
         }
       float targetReputation = Float.valueOf( getLastReputation
           ( idTarget ) );
         totalRep += targetReputation;
         return totalRep;
     }
}
```

7.4 REGRET

REGRET [27] is a reputation model that considers three reputation values, one for each considered dimension: an individual dimension, a social dimension and an ontological dimension. The three reputation values are calculated from a set of impressions gathered by the entities. These impressions are about a subject and a target, and map to what we call claims.

The individual dimension calculates a so-called subjective reputation value by using impressions of the agent about the target agent, as follows:

$$R_{a \to b}(subject) = \sum \rho(t, t_i) \cdot W_i$$

where a is the source entity, b is the target entity, W_i is the claim value in the range $[-1, 1]$, and ρ is a function that gives recent impressions a higher weight.

For the social dimension, the model considers that agents belong to groups, denoted by $\mathcal{A}$, $\mathcal{B}$, etc, and calculate the reputation at the group level, considering the impressions about each agent of the group. In particular, the model considers:

$$\begin{aligned} R_{a \to \mathcal{B}}(subject) &= \textstyle\sum_{b_i \in \mathcal{B}} w \cdot R_{a \to b_i}(subject) \\ R_{\mathcal{A} \to b}(subject) &= \textstyle\sum_{a_i \in \mathcal{A}} w \cdot R_{a_i \to b}(subject) \\ R_{\mathcal{A} \to \mathcal{B}}(subject) &= \textstyle\sum_{a_i \in \mathcal{A}} w \cdot R_{a_i \to \mathcal{B}}(subject) \end{aligned}$$

where w are weights that must sum up 1. The final reputation value consists of a weighted sum of all the previous values.

The model also considers an ontological dimension, where a subject (or context) might be decomposed into other subjects, which allows generalizing a reputation value for a new subject from weighting the contributing existing subjects.

In order to implement this model in our framework, we make some slight simplifications. The most important one is that we do not consider the ontological dimension, because contexts relationships are not supported in the framework. Also, in order to simplify calculations and show a more clear code, we assume a uniform distribution of weights across impressions and we do not consider reliability of the reputation values.

The coarse-grained steps for the implementation are the following:

- Inheriting from *DistKevReputableEntity*, invoking the method *makeClaim* upon receiving a message, which simulates the impressions.
- Set the group to which each entity belongs.
- The reputation engine must retrieve the impressions of all entities to compute the different reputation values, and the groups to which each entity belongs.

The code for the consoles is the same as the one depicted in Listing 6, except that we bind another reputation engine in the *start()* method. The reputation engine implements the formula explained earlier, as shown in Listing 8.

Listing 8: Binding Reputation Engine and Console

```
public class RegretReputationModel extends ReputationEngine<
    Float>
{
      @Override
    public Float compute(String context, String idTarget,
        String idSource) {

        //1) Calculate subjective reputation
        List<ReputationStatementInfo> claims = getClaims( "
            CleanWords", idSource, idTarget );
        float subjectiveReputation = 0.0f;
        float totalClaims = claims.size();
        double currentTime = (double) Calendar.getInstance().
            getTime().getTime();
        for( ReputationStatementInfo rs : claims )
        {
            float claimVal = Float.valueOf( rs.getClaim().
                getValue() );
            double claimTimeStamp = (double) rs.getTimeStamp();
            subjectiveReputation += (claimVal / totalClaims) *
                (claimTimeStamp / currentTime);
        }

        //2) Now, retrieve all the claims that idSource made
            about any entity in the same group as idTarget
        String groupTarget = getParam( idTarget, "group" );
        List<ReputationStatementInfo> targetGroupClaims =
            getClaimsFromSource("CleanWords", idSource);
        float targetGroupReputation = 0.0f;
        currentTime =  Calendar.getInstance().getTime().getTime
            ();
        for( ReputationStatementInfo rs: targetGroupClaims )
        {
            //We don't want to consider claims about the target
                itself
            if ( !idTarget.equals( rs.getTarget() ))
            {
                String g = getParam(rs.getTarget(), "group");
                if (groupTarget.equals(g)) {
                    float claimVal = Float.valueOf(rs.getClaim
                        ().getValue());
                    double claimTimeStamp = (double) rs.
                        getTimeStamp();
                    targetGroupReputation += (claimVal /
                        totalClaims) * (claimTimeStamp /
                        currentTime);
                }
            }
```

```
        }

        //3) Now, retrieve all the claims that any entity in
            the same group as idSource made about idTarget
        String groupSource = getParam( idSource, "group" );
        List<ReputationStatementInfo> sourceGroupClaims =
            getClaimsAboutTarget( "CleanWords", idTarget );
        float sourceGroupReputation = 0.0f;
        currentTime =  Calendar.getInstance().getTime().getTime
            ();
        for( ReputationStatementInfo rs : sourceGroupClaims )
        {
            //We don't want to consider claims from the source
                itself
            if ( !idSource.equals( rs.getSource() ))
            {
                String g = getParam(rs.getSource(), "group");
                if (groupSource.equals(g)) {
                    float claimVal = Float.valueOf( rs.getClaim
                        ().getValue() );
                    double claimTimeStamp = (double) rs.
                        getTimeStamp();
                    sourceGroupReputation += (claimVal /
                        totalClaims) * (claimTimeStamp /
                        currentTime);
                }
            }
        }

        return new Float( subjectiveReputation +
            targetGroupReputation + sourceGroupReputation );
}
```

8 Discussion

This section describes the experiment that we carry out in order to measure the performance overhead that the framework entails, as well as the amount of work that developers need to invest during the implementation of the models. We also enumerate some shortcomings and technical challenges that we faced and that need to be overcome to obtain the most out of a trust-aware and self-adaptive framework.

8.1 Experimental Results

We design an experiment to measure the performance overhead of the trust framework and the reconfiguration mechanism. The application used for the experiment is the one explained in Section 7. In order to ignore network latency, both consoles are executed

on the same platform, which is a 2010 Macbook Pro Intel Core 2 Duo, with 4GB 1067 MHz DDR3 RAM.

The experiment is as follows. First, we measure the time elapsed between the time the first console sends a message and the second console shows it, without any trust or reputation involved. Then, for each trust or reputation model considered in Section 7, we measure this same time. In order to account for the computation engines, the receiver console shows the text only after it has received an update of the sender console trust or reputation. Each measure is actually an average of 100 individual measures to provide more statistically meaningful results, which are depicted in Fig. 5. We can observe that there is a small overhead when using the framewok, although this overhead comes in terms of micro-seconds. The least overhead comes from Marsh's model, whereas the greatest comes from REGRET, something expected given its more complex computation engine.

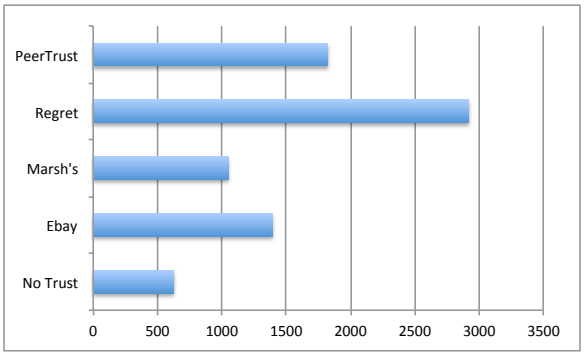

Figure 5: Execution Time (measured in microseconds)

The amount of work that takes for developers to implement the models is similar for all the models, as shown in Table 2.

Table 2: Amount of Framework-related Activities

	eBay	**Marsh's**	**PeerTrust**	**REGRET**
#inheritance	2	2	2	2
#invocations	3	4	4	6
#overriding	2	3	2	2
#configFiles	1	2	1	1
#compDeployed	3	3	2	2

As explained in Section 7, each model requires inheriting from two framework classes, the class that determines the type of the entity, and the class that implements the model or the engine. The number of method invocations go hand in hand with the complexity of the model. Thus, REGRET requires up to 6 method calls whereas eBay only requires 3. The number of methods that need to be overridden is similar in all the models, although it is higher in Marh's model because it needs to implement

the threshold value. All models require at least one configuration file with the self-adaptation policy, whereas Marh's require another one for setting the initial subjective factors of the entities. The deployment changes slightly, as in the case of centralized models (like Marsh's and centralized reputation models like eBay's), three components must be deployed, whereas in distributed reputation models only two are required[13].

As a conclusion, the framework entails negligible overhead (in the order of micro-seconds) and does not require a lot of work to implement well-known, existing models. This means that the benefits of adopting the framework are quite high considering the work that the implementation requires (see Table 2) or even in terms of execution time, as shown in Fig. 5. We advocate that these results make the adoption of the framework appealing.

8.2 Challenges

We have learned that this kind of integration must overcome several challenges. In our view, one primary challenge is building a robust identity management system in order to uniquely identify trust and reputation entities, and to allow access at any moment to these identities. In our current implementation, we build upon the reflection layer of Kevoree so as to provide such identities. However, it would be desirable to keep track of entities that disappear and re-appear again in dynamic environments, which is something we do not tackle at the moment.

Second, more research on declarative reconfiguration policies is required. Current models@run.time platforms lack a usable mechanism to specify reconfiguration policies. Kevoree provides Kevscript instructions, which become cumbersome for advanced reconfigurations. We have provided a basic format to represent these policies, but it may fall short of expressiveness as the complexity of scenarios increases.

We also find that models@run.time platforms should provide a great deal of usable low-level services in order to monitor certain aspects of the system, like the consumed resources by each component, the latency of communications, or the response times, because this information might be key to building robust trust and reputation models. Factor producer entities could use these services to monitor different aspects of the entities and feed the trust or reputation engine.

9 Conclusion and Future Work

In this paper we have developed a trust and reputation framework that allows implementing a wide range of trust and reputation models. The framework has been implemented on the top of a self-adaptive platform, which enables the use of trust and reputation information in order to make reconfiguration decisions. We have shown that the framework barely entails overhead and that the small amount of extra work for developers pays off given the interesting opportunities brought by the framework.

Despite the huge amount of trust and reputation models proposed in the literature, we have found that by using only some core concepts (embodied in trust and reputation

[13] In practice, three components should be deployed in order to test the reputation engines of PeerTrust and REGRET due to their consideration for groups and credibility.

metamodels), it is possible to represent a wide range of them. This happens because the differences among models are often due to the application context where the models are proposed, rather than in their dynamics, which turn out to be similar in most cases.

As future work, we intend to address the following issues. First, an empirical validation of the framework usability is required. In particular, we intend to observe other developers while they implement different trust and reputation models. In this same direction, we are interested in improving the usability of the API by supporting the configuration-based (e.g. XML) development of trust and reputation models, by reducing the amount of code that developers must write and providing a higher-level interface to the trust and reputation management. We also plan to test the framework in a more elaborated scenario concerning smart grids, where multiple heterogeneous entities interact. Finally, we would like to address other types of trust models, like propagation models, where new trust relationships are derived from existing ones by exploiting trust transitivity.

Acknowledgements

This work has been partially supported by the European Training Network NeCS (H2020-MSCA-ITN-2015- 675320) and by the Spanish Ministry of Economy and Competitiveness through the project PERSIST (TIN2013-41739-R). The second author is supported by the Ministery of Economy of Spain through the Young Researchers Programme: project PRECISE (TIN2014-54427-JIN).

The authors wish to thank Benoit Baudry, Jean-Emile Dartois, Erwan Daubert and François Fouquet for their invaluable feedback and support during this research.

References

[1] Gordon Blair, Nelly Bencomo, and Robert B. France. Models@ run.time. *Computer*, 42(10):22–27, 2009.

[2] William Conner, Arun Iyengar, Thomas Mikalsen, Isabelle Rouvellou, and Klara Nahrstedt. A trust management framework for service-oriented environments. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 891–900, New York, USA, 2009. ACM.

[3] C Crapanzano, F Milazzo, A De Paola, and G.L Re. Reputation Management for Distributed Service-Oriented Architectures. In *Proceedings of the Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop (SASOW'10)*, pages 160–165, 2010.

[4] Randy Farmer and Bryce Glass. *Building Web Reputation Systems*. Yahoo! Press, USA, 1st edition, 2010.

[5] François Fouquet, Olivier Barais, Noël Plouzeau, Jean-Marc Jézéquel, Brice Morin, and Franck Fleurey. A Dynamic Component Model for Cyber Physical Systems. In *Proceedings of the 15th International ACM SIGSOFT Symposium on*

Component Based Software Engineering, pages 135–144, Bertinoro, Italie, July 2012.

[6] John C. Georgas, André van der Hoek, and Richard N. Taylor. Architectural Runtime Configuration Management in Support of Dependable Self-adaptive Software. *SIGSOFT Software Engineering Notes*, 30(4):1–6, May 2005.

[7] Carlo Ghezzi. The Fading Boundary between Development Time and Run Time. In *Proceedings of the Ninth IEEE European Conference on Web Services (ECOWS'11)*, page 11, Sep 2011.

[8] Haouas Hanen and Johann Bourcier. Dependability-Driven Runtime Management of Service Oriented Architectures. In *4th International Workshop on Principles of Engineering Service-Oriented Systems (PESOS'12)*, pages 15–21, Zurich, Switzerland, June 2012.

[9] P. Herrmann and H. Krumm. Trust-adapted enforcement of security policies in distributed component-structured applications. In *Proceedings of the 6th IEEE Symposium on Computers and Communications*, pages 2–8, 2001.

[10] Peter Herrmann. *Trust Management: First International Conference on Trust Management (iTrust'03)*, volume 2692 of *LNCS*, chapter Trust-Based Protection of Software Component Users and Designers, pages 75–90. Springer Berlin Heidelberg, May 2013.

[11] Audun Jøsang, Roslan Ismail, and Colin Boyd. A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 43(2):618–644, March 2007.

[12] Rolf Kiefhaber, Florian Siefert, Gerrit Anders, Theo Ungerer, and Wolfgang Reif. The Trust-Enabling Middleware: Introduction and Application. Technical report, Institut für Informatik Universität Augsburg, March 2011.

[13] L. Klejnowski, Y. Bernard, J. Hähner, and C. Müller-Schloer. An Architecture for Trust-Adaptive Agents. In *Proceedings of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'10)*, pages 178–183. IEEE, 2010.

[14] L. Luca, D. Pierpaolo, B. Riccardo, B. Stephen, and B. Andrew. Enabling Adaptation in Trust Computations. In *ComputationWorld '09: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pages 701–706. IEEE, 2009.

[15] Stephen Marsh. *Formalising Trust as a Computational Concept*. PhD thesis, University of Stirling, April 1994.

[16] Matt Blaze and Joan Feigenbaum and Jack Lacy. Decentralized Trust Management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.

[17] Francisco Moyano, Benoit Baudry, and Javier Lopez. Towards trust-aware and self-adaptive systems. In Carmen Fernandez-Gago, Isaac Agudo, Fabio Martinelli, and Siani Pearson, editors, *Proceedings of the 7th IFIP WG 11.11 International Conference on Trust Management (IFIPTM'13)*, volume 401 of *AICT*, pages 255–262, Malaga, Jun 2013. Springer, Springer.

[18] Francisco Moyano, Carmen Fernandez-Gago, and Javier Lopez. A conceptual framework for trust models. In Simone Fischer-Hübner, Sokratis Katsikas, and Gerald Quirchmayr, editors, *Proceedings of the 9th International Conference on Trust, Privacy & Security in Digital Business (TrustBus'12)*, volume 7449 of *Lectures Notes in Computer Science*, pages 93–104, Vienna, Sep 2012. Springer Verlag.

[19] Francisco Moyano, Carmen Fernandez-Gago, and Javier Lopez. A framework for enabling trust requirements in social cloud applications. *Requirements Engineering*, 18:321–341, Nov 2013.

[20] S. Phoomvuthisarn, Yan Liu, and Jun Han. An Architectural Approach to Composing Reputation-Based Trustworthy Services. In *Proceedings of the 21st Australian Software Engineering Conference (ASWEC'10)*, pages 117–126, 2010.

[21] Harald Psaier, Lukasz Juszczyk, Florian Skopik, Daniel Schall, and Schahram Dustdar. Runtime Behavior Monitoring and Self-Adaptation in Service-Oriented Systems. In *Proceedings of the IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO'13)*, volume 0, pages 164–173, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[22] Sarvapali D Ramchurn, Dong Huynh, and Nicholas R Jennings. Trust in multi-agent systems. *The Knowledge Engineering Review*, 19(01):1–25, April 2005.

[23] Lars Rasmusson and Sverker Jansson. Simulated social control for secure internet commerce. In *Proceedings of the 1996 workshop on New security paradigms*, NSPW '96, pages 18–25, New York, NY, USA, 1996. ACM.

[24] Paul Resnick, Richard Zeckhauser, John Swanson, and Kate Lockwood. The value of reputation on ebay: A controlled experiment. *Experimental Economics*, 9(2):79–101, Jun 2006.

[25] Paul Robertson and Robert Laddaga. Adaptive Security and Trust. In *Proceedings of the 6th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'12)*, pages 55–60. IEEE Computer Society, 2012.

[26] Paul Robertson, Robert Laddaga, and Mark H. Burstein. Trust and Adaptation in STRATUS. In *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom'13)*, pages 1711–1716. IEEE, 2013.

[27] Jordi Sabater and Carles Sierra. REGRET: Reputation in Gregarious Societies. In *Proceedings of the Fifth International Conference on Autonomous Agents*, AGENTS '01, pages 194–195, New York, NY, USA, 2001. ACM.

[28] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), Feb 2006.

[29] Quang-Anh Nguyen Vu, Salima Hassas, Frederic Armetta, Benoit Gaudou, and Richard Canal. Combining Trust and Self-Organization for Robust Maintaining of Information Coherence in Disturbed MAS. In *Proceedings of the Fifth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'11)*, pages 178–187. IEEE, 2011.

[30] Li Xiong and Ling Liu. PeerTrust: Supporting Reputation-Based Trust for Peer-to-Peer Electronic Communities. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):843–857, July 2004.

[31] Zheng Yan and C. Prehofer. Autonomic Trust Management for a Component-Based Software System. *IEEE Transactions on Dependable and Secure Computing*, 8(6):810–823, 2011.

[32] Zheng Yan, Peng Zhang, and Teemupekka Virtanen. Trust evaluation based security solution in ad hoc networks. In *Proceedings of the Seventh Nordic Workshop on Secure IT Systems, (NordSec'03)*, October 2003.

A Implementation Listings

Listing 9: TrustEntity Component

```
@ComponentType
public class TrustEntity<T, F>
{
    @Param(defaultValue = "both")
    private String role;

    @Param(defaultValue = "MyContext")
    private String trustContext;

    @Param(defaultValue = "MyGroup")
    private String group;

    @Param(defaultValue = "0")
    private String bootstrappingTrustValue;

    @Param
    private String subjectiveFactorsFilePath;

     @Output
    private Port requestTrustUpdate;

    @Output
    private Port initTrustRelationships;
```

```
    @Output
    private Port addFactor;

   @KevoreeInject
    private Context context;

    private String uid;

   @Start
    protected final void start()
    {
        uid = context.getInstanceName() + "@" + context.
            getNodeName();
         initTrustRelationships.send(trustContext + "." + uid +
              "." + bootstrappingTrustValue);
        storeSubjectiveFactors();
    }
}
```

Listing 10: Initialization of Trust Entities in Trust Model Component

```
@ComponentType
public class TrustModel<T,F> implements IComputationEngine
{
   @Input
    protected final void initializeTrustRelationships( Object
        request )
    {
        StringTokenizer st = new StringTokenizer( request.
            toString(), "." );
        String context = st.nextToken();
        String idTrustor = st.nextToken();
        String bootstrappingTrustValue = st.nextToken();

        HashMap<String, List<String>> trustees =
                GetHelper.getTrusteesInstanceName(model.
                    getCurrentModel().getModel(), context,
                    idTrustor );
        List<String> idTrustee = new ArrayList<String>();

        for (String nodeName: trustees.keySet()) {
            for ( String compName : trustees.get( nodeName ))
            {
                idTrustee.add( compName + "@" + nodeName );
            }
        }

        for (String t : idTrustee) {
            long ts = Calendar.getInstance().getTime().getTime
```

```
                ();
            addTrustRelationship( context, idTrustor, t,
                bootstrappingTrustValue, ts );
        }
    }

    public T compute(String context, String idTrustee, String
        idTrustor)
    {
        return null;
    }
}
```

Listing 11: Using the EMF API to add Trust Relationships

```
private void addTrustRelationship( String context, String
    idTrustor, String idTrustee, String initialValue, long
    timeStamp )
{

   Trustee trustee = trustModel.findTrusteesByID( idTrustee );
   if ( trustee == null )
   {
         trustee = factory.createTrustee();
         trustee.setIdTrustee( idTrustee );
         trustModel.addTrustees( trustee );
     }

   //Creation of the rest of trust relationships elements
}
```

Listing 12: TrustModel

```
private void addTrustRelationship( String context, String
    idTrustor, String idTrustee, String initialValue, long
    timeStamp )
{

   protected final String getFactorValue(String context, String
       name, String uidTarget)
    {
        for( Factor f : trustModel.getFactors() )
        {
            if (context.equals(f.getContext()) &&
               name.equals(f.getName()) &&
               uidTarget.equals(f.getIdTarget()))
            {
                return f.getValue().getValue();
            }
        }
```

```
        return null;
    }

    public T compute(String context, String idTrustee, String
        idTrustor)
    {
        return null;
    }

    protected T computeThreshold(String context, String
        idTrustee, String idTrustor)
    {
      return null;
    }
}
```

Listing 13: CentralReputableEntity Component

```
@ComponentType
public class CentralReputableEntity<T> implements IClaimSource
    {

    @Param(defaultValue = "MyContext")
    private String trustContext;

    @Param(defaultValue = "MyGroup")
    private String group;

    @Output
    private Port sendClaim;

    @Output
    private Port requestReputation;

    private String uid;

    protected void reputationReceived( String target, T newVal
        ) { }
}
```

Listing 14: ReputationManager Component

```
@ComponentType
public class ReputationManager<T> implements IComputationEngine
{
   //...

   protected final List<String> getClaimsValues( String context
       , String name, String target ) {
        List<String> claims = new ArrayList<String>();
```

```
        for (ReputationStatement rs : repRoot.getStatements())
        {
            if (rs.getContext().equals( context ) && rs.
                getTarget().getIdTarget().equals( target ))
            {
                for(Claim claim : rs.getClaim())
                {
                    if(claim.getName().equals(name))
                    {
                        claims.add( claim.getClaimValue().
                            getValue() );
                    }
                }
            }
        }
        return claims;
    }

    public T compute( String context, String idTarget, String
        idSource )
    {
        return null;
    }

    //...
}
```

Listing 15: DistReputableEntity Component and Reputation Engine Initialization

```
public class DistReputableEntity<T> implements IClaimSource {

    @Param(defaultValue = "MyContext")
    private String trustContext;

    @Param(defaultValue = "MyValue")
    private String group;

    @KevoreeInject
    private Context ctx;

    @Output
    private Port requestClaim;

    private String uid;
    private ReputationEngine<T> reputationEngine;

    @Start
    public void start( ReputationEngine repEngine )
    {
        uid = ctx.getInstanceName() + "@" + ctx.getNodeName();
```

```
        reputationEngine = repEngine;

        //...

    }

    //...
}
```

Listing 16: ReputationEngine Class

```
protected final List<ReputationStatementInfo>
    getClaimsFromSource(String name, String idSource)
{
        List<ReputationStatementInfo> claims=new ArrayList();
        for( ReputationStatementInfo cInfo :
            reputationStatements )
        {
            if( name.equals(cInfo.getClaim().getName()) &&
                cInfo.getSource().equals( idSource ))
            {
                claims.add( cInfo );
            }
        }
        return claims;
    }
```

Listing 17: Reconfiguration Rules Processing

```
@ComponentType
public class DistReputableEntity<T> implements IClaimSource {

   @Start
    public void start( ReputationEngine repEngine, String
        fileName )
    {
        se = new ScriptEngine( model );
        rre = new ReputationRulesEngine( fileName, se );
    }

   private void computeReputation( String idTarget, boolean
       reconfigure, List<ReputationStatementInfo> rsInfo )
    {
        T res = reputationEngine.compute( model, trustContext,
            idTarget, uid, rsInfo, this );

        if ( reconfigure )
        {
            rre.executeRules( model, idTarget, res.toString() )
                ;
```

```
        }

        //...
    }
}
```

Listing 18: ScriptEngine: Remove Component

```
private void removeComponent( String idComponent )
 {
   StringTokenizer st = new StringTokenizer( idComponent, "@" )
       ;
    String instance = st.nextToken();
    String node = st.nextToken();

    String script = "remove " + node + "." + instance;
    model.submitScript(script, new UpdateCallback() {
    @Override
     public void run(Boolean applied) {

      }
   });
}
```