SCALE

A language for dynamic composition of heterogeneous services
## Contents

1 Introduction ................................................................................................................. 4  
   1.1 Heterogeneous Service Landscape ......................................................................... 5  
   1.2 Composition of heterogeneous services ............................................................... 7  

2 Guiding Concepts and Central Objectives ................................................. 8  
   2.1 Heterogeneous service environments ................................................................... 8  
   2.2 SOA at its heart .................................................................................................. 9  
   2.3 Technology agnostic composition through common language elements and abstraction ...................................................................................................... 10  
   2.4 Dynamic service selection .................................................................................. 10  
   2.5 Asynchronous by default – control flow by constraints ....................................... 11  
   2.6 Constraint based and abstract composition design ............................................. 11  
   2.7 Events and messages ......................................................................................... 11  
   2.8 Extensible to future service technologies .......................................................... 12  
   2.9 Scalable language – simplicity in composition ................................................... 12  
   2.10 Graphical and textual syntax ............................................................................ 12  
   2.11 Aspect Orientation ........................................................................................... 13  
   2.12 SCALE feature summary ................................................................................. 13  

3 Reference architecture of a SCALE Engine .................................... 13  

4 Key Elements ......................................................................................................... 15  
   4.1 Syntax presented in this paper ............................................................................ 15  
   4.2 Skeletons ............................................................................................................ 15  
   4.3 Actions ............................................................................................................... 15  
   4.4 Flexible Constraints ........................................................................................... 33  
   4.5 Scopes ................................................................................................................ 35  
   4.6 Events ............................................................................................................... 38  
   4.7 Utilizing external elements and definitions ......................................................... 46  

5 Data Handling ....................................................................................................... 47  
   5.1 Shared State ...................................................................................................... 47  
   5.2 Locality ............................................................................................................ 47  
   5.3 Annotations ....................................................................................................... 48  
   5.4 Data Type System .............................................................................................. 49  
   5.5 Optional External Type Extensions .................................................................... 49  

6 Heterogeneous Composition Examples ............................................ 50  
   6.1 Request-Response service usage ....................................................................... 51  
   6.2 Persistent Processes ............................................................................................ 51  
   6.3 Composition of SIP services ............................................................................. 52  
   6.4 Composition and control of SIP Service Chains .............................................. 55  
   6.5 Heterogeneous composition of SIP and Web Services ..................................... 57  
   6.6 Practical considerations of technology hiding .................................................... 58  

7 Aspect Oriented Programming in SCALE ............................................. 59
8 SCALE at Simple and Expert level .....................................................60
9 Conclusion and Outlook .....................................................................60
10 Glossary................................................................................................60
11 References............................................................................................61
1 Introduction

The service landscape today consists of many different technologies used to implement services. What we observe today is the result of a historical development process aiming to find solutions to concrete technical problems within business cases. The traditional telecommunication industry for example had other demands than for example enterprise solutions or the public internet. While originally these were separate domains, each of them produced an own portfolio of service technologies. As a result we see a service environment which is to a great extent heterogeneous.

Convergence between the traditionally separated IT/internet, enterprise and telecommunication industries can already be observed for a while, but this development gains significant momentum due to current and future market demands. The internet of things, the expectation of 50 billion devices and the exploding popularity of smartphones are only a few examples, which mark an inflationary growth of assets with increased and at the same time highly diverse communication demands. In order to accommodate them, networks will be needed, which not only connect devices to each other, but which provide a dynamic service infrastructure as backend. In this respect it is also not enough to provide a few very attractive mass services. The future scenario as outlined above requires diversification and individually customized services. Service Composition is one answer to this need and an essential technique for service creation and integration in future service networks.

In recent years technologies, which allow orchestrating services into applications based on modeling of business processes and automatic workflows, have gained significant momentum and wide acceptance. Architectural concepts like Service Oriented Architecture (SOA) are well established. Technologies like for example SOAP/web services and the Web-Services Business Process Execution Language (WS-BPEL) did emerge in order to implement a SOA. They are deployed in many practical use-cases. In parallel extensive research did explore their capabilities and also their limitations.

Ericsson has done extensive research in recent years targeting flexible service environments that apply SOA principles to telecommunication networks and at the same time integrate telecommunication services with other service worlds. One result of this research was, that existing and popular languages for defining compositions like BPEL are very workflow centric, while many practical use-cases that include telecommunication services are highly event driven and often also require stateful implementation. These experiences resulted in a strong demand for a new language that is able to efficiently specify compositions in heterogeneous service environments and that is also relatively easy to understand and use. This paper provides an insight into SCALE, a new composition language targeting heterogeneous compositions within heterogeneous service environments.

The paper discusses which semantic elements a language should have, which is made for specifying heterogeneous service compositions within multiple
application domains. The practical use of this language is shown in examples. In order to do so this paper also proposes an graphical syntax as one possible embodiment of the semantic elements of the language.

At the time of writing this paper the language is still in an early phase towards standardization. We are aiming at OASIS as standardization organization due to its high visibility and relevance for both the IT industry and the telecommunication industry.

1.1 Heterogeneous Service Landscape

In this chapter we will provide an overview of some of the most relevant and most widely used service technologies in order to outline central differences and to motivate the approaches and elements of the SCALE composition language.

1.1.1 Request-Response style service consumption

This is a widespread way of service usage. A user issues a request to a service. The service processes the request and finally provides a result back to the user. Usually the service can be stateless and the client is waiting for the result before proceeding. This means the coordination between client and service is relatively simple. The usage of web services often follows this scheme.

1.1.2 Business process and workflow modeling

When web services are orchestrated, a central execution engine takes the role of client towards a number of basically independent services. Languages like for example WS-BPEL are used in order to specify and control this process of orchestration. BPEL allows to model workflow-like business processes. When executing the business process, the execution engine binds and invokes web services in order to integrate their function into the overall application.

The main considerations for an orchestration model are:

- which services shall be used as constituent components,
- what execution order shall be applied to constituent services and
- how does data flow between the services?

Figure 1 shows the relation between user, business process and constituent web services. The user is served by means of instantiating web services as defined and controlled by a business process.
Constituent web services are integrated into the orchestration following a request-response usage scheme. However, a service may in turn be orchestrated from further web services. Therefore, an overall orchestration can get a hierarchical tree structure with the business process at central controlling user at the root.

1.1.3 Asynchronous responses and inter-work of parallel processes

The response to a service request might be asynchronous in the sense, that the client is not blocked until a response is received from the service. Further operations might be executed by the client before the result of the service request is available. It is still a request-response scheme, but the responses will come asynchronously. AJAX is an example for a service technology where this kind of service usage is widely applied.

The challenge is the coordination of the incoming asynchronous responses. Other operations of the client might for example require data expected in the response. This bears the danger of errors from race conditions. Frequently state machines and timers are used in order to coordinate.

One step ahead but with similar challenges is the case that the services is in fact a persistent process. The inter-work between the client process and the service process might be characterized by agile exchange of messages. Such a constellation usually requires considerable coordination effort by means of for example state machines.

1.1.4 Service chaining in end-to-end sessions

Telecommunication service networks are based on the end-to-end communication sessions between two users. Control signaling implemented by end-to-end protocols is propagated through the network along the session path in both directions. Today the actual payload is separated from the
signaling and the path becomes more a virtual one. Nevertheless the concept is still the foundation of telecommunication networks.

Services in this scenario are self contained unique entities residing on the session path. This path becomes a logical chain through from a user at one end-point through a number of services in a certain order towards another user at the other endpoint. These services listen to the end-to-end signaling in order to decide when to get active and when to execute a certain function. This can result for example in further signaling being issued to the chain. Note, that in this respect the position of a particular service instance on the logical chain is significant.

![Telecommunication Session Control](e.g. IMS)

**Figure 2** SIP Service Chaining

Services like this are frequently stateful as they need to coordinate with a persistent session. From a single service’s viewpoint the observed signaling on the session chain is asynchronous. The coordination and inter-work between services and between users and service is usually guided by detailed standards and well defined protocols and APIs.

### 1.2 Composition of heterogeneous services

The previous chapters provided a brief overview over some concepts and technologies in today’s service landscape including orchestration and composition. There are considerable differences in the way services and service users interact, the topology of the usage scenario and roles taken by services and service users. The complexity range starts with stateless and blocking inter-work and it ends at persistent processes following complex protocols.

A heterogeneous composer needs to be enabled to operate in such a diverse and heterogeneous environment. This means it needs to be able to utilize services from this multitude of different technical approaches as constituent components within a single composition. An arbitrary mix of technologies shall be possible and controllable in a preferably intuitive way. Furthermore the overall composition itself shall be useable as service for a multitude of different users.

Comprehensive and nevertheless intuitive control of this kind of envisioned composition is the sole purpose of a composition language in general. This is
what SCALE is designed for. The general approach of SCALE is to provide abstract concepts that fit to all service technologies in scope. In SCALE a set of common elements and concepts exists, which comprehensively provides sufficient control.

Other languages and development environments exist today that have similar targets and could be used to build service applications and compositions. It is of course possible to create an application based on a general purpose programming language like for example Java. External service components can be called from the Java program in order to integrate them. However, a general purpose language requires the developer to take care of many low-level tasks when integrating. This is not ideal for rapid development, the flexible creation of variants of an application and it demands low-level testing.

A development environment and a language that is specific to the domain of creating service compositions allow the developer to work on higher abstraction level. The standard low level tasks would be offloaded into the composition execution engine. WS-BPEL is such a domain-specific language and it became very popular in recent years. However, WS-BPEL is mainly made for specifying compositions that have a workflow nature and it is focusing on specific service technologies like SOAP Web Services. Telecommunication services with their event-driven nature and often stateful implementation do not fit well into workflow based compositions with web service interfaces.

These two examples show the need of either creating a new language for the composition domain or alternatively extending existing languages and their respective tooling. This paper outlines a new language proposal that has the desired properties required for heterogeneous compositions.

2 Guiding Concepts and Central Objectives

This chapter introduces the central objectives and properties of the SCALE composition language and outlined already the key concepts how they are reached.

2.1 Heterogeneous service environments

The SCALE language allows the creation of compositions with constituent component services based on a broad range of technologies. The targeted service domains comprise SOA, the Internet in general, enterprise services and telecommunication services. The support of service technologies includes for example web services as today’s most prominent service technology for SOA. Internet based applications are supported by means of for example Ajax and Http. Services from the enterprise domain can be utilized for example through ESBs and finally telecommunication services based on IN/CAMEL and on IMS/SIP technology are included.
Although these technologies might significantly differ the SCALE compositions can compose them together, thus integrate them. SCALE allows being comprehensive in support for services of very different kinds.

2.2 SOA at its heart

The SCALE language is designed following the key concepts of a service-oriented architecture. This includes for example the notion of a service to be a self contained unit, which provides its functionality through dedicated interfaces.

Furthermore the concept of a service broker and a publish-find-bind scheme in the interaction between users, brokers and services is deployed. Services publish their function at a service brokers. Users in need of a certain function find services through the broker and finally bind the service in order to integrate it into an overall application. Late binding and loose coupling are key concepts in this kind of service environment and also a key property of the service usage implied by SCALE.
2.3 Technology agnostic composition through common language elements and abstraction

The list of considered services and service domains presented in Chapter 2.1 is already a comprehensive collection of today’s widely used thus relevant service technologies. Nevertheless the language is designed to be agnostic of the service implementation.

SCALE contains a number of elements, which allow interacting with a great variety of differently implemented services in a uniform way. For example external actions in the constituent service usage are exposed by events, external data is exposed by shared state variables and services are abstracted through a high level service description. The interaction with external services of any type or technology is broken down to the usage of a few core elements and concepts, which are the same for any type of service.

Especially the abstraction of services plays a central role in hiding irrelevant technological details. All services are described on by abstract properties. Services are exposed to a developer by means of these abstract properties. The idea is that it does not matter how a service is implemented as long as it provides a certain needed property.

2.4 Dynamic service selection

In SCALE the concrete service to be used as constituent component of the composition is not pre-determined by the composition developer. The developer only specifies constraints which express the required abstract properties of a constituent service. It is the composition execution engine at run-time, which dynamically select a service that fulfills the requirements.

This dynamic service selection leads to a very loose coupling and it goes one step ahead of late binding. For simple late binding it is pre-determined which service to use although the actual service is only needed at run time. Dynamic selection does not only perform the binding at run-time, but also the decision which services are used at all.

Also here the abstract description of services plays a central role. The abstract properties used to describe the available services are also used in order to phrase conditions and requirements. For example, the geographical location of a user might be needed. In order to get this information the composition developer would use a constraint that requires a service described with the property “user location”. This constraint is evaluated by the composition execution engine just in time in order to find a suitable service. In general this can be a different service every time the composition is executed, but this does not matter as long as the service fulfils the constraint, thus provides the needed data. That was the save.
2.5 Asynchronous by default – control flow by constraints

All elements of a composition which is specified by the SCALE language can be executed in any order or in parallel unless otherwise it is explicitly stated or implicitly implied by dependencies. This leaves a lot of freedom to the engine that is executing the composition with regards to automatic optimizations gained from a dynamically decided order of execution.

In order to guide the engine the developer provides information that needs to be taken into account by means of constraints. This can for example be a data dependency implied by one constituent service expecting output of another service as input. The language will provide means to model the important dependencies. Thus, the control flow of a SCALE composition is constraint driven.

2.6 Constraint based and abstract composition design

As already indicated in chapters 2.4 and 2.5, constraints are a central element in SCALE. Constraints are used to describe the abstract properties a service has to fulfill for being suitable as constituent component within a composition. Thus constraints are used for dynamic service selection based on abstract service descriptions. Furthermore constraints model dependencies between services and guide the composition execution engine to execute the constituent services in a certain order.

This means both, the set of services to be used and the execution flow are both determined by constraints and decided dynamically while execution of the composition. Furthermore, constraints are data and can change dynamically while execution, also the selected set of services and the execution flow can be highly dynamic.

2.7 Events and messages

Another central concept in SCALE will be events and in particular asynchronous events. Events are the primary trigger for composition and execution. External communication with services is exposed by means of events. Events can bear data, thus they can be used as messages being exchanged between various parts of a composition and between the composition and external services and users.

Events in SCALE accommodate the often asynchronous nature of interaction between services which can be persistent and independent processes rather than stateless and simple. Especially telecommunication services are often self-contained and stateful entities within a communication session.
2.8 Extensible to future service technologies

The support for a certain service technology is modular. Based on the technology agnostic composition core, which is executing a composition defined in SCALE, the envisioned execution environment will be able to define technology specific handling through well encapsulated modules. Adding new modules representing yet another service technology is a key concept of the overall composition execution environment. It might even be possible in many cases to utilize service from a new service technology without changing existing compositions. This allows being future proof.

Next to the modular addition of new service technologies there are other parts of the language designed to be extensible. One example for this are data-types and data annotations. New data-types and respective data operations can be defined by means of optional modules. Furthermore, data annotations can be used in order to convey additional information with the data.

2.9 Scalable language – simplicity in composition

One major goal of this composition language is to enable un-experienced developers specifying composition without long training and a steep learning curve needed. This fosters rapid development cycles targeting for example development for the long tail market with its specific demands on low cost development of variants. At the same time the language shall allow experienced developers to control many aspects of the composition in detail. The language will therefore specify at least two experience levels. All these levels will operate with the same basic elements. The entry level language will work to a great extent with default behavior of the language components, while the expert level of the language allows extensive deviation from the default behavior and advanced control. However, this paper provides an overview of the key concepts of the full language. A separation in simple subsets is not described in detail here.

2.10 Graphical and textual syntax

The semantic language elements shall have both, a textual and a graphical representation. Both are equally representing the semantic elements and concepts of the SCALE language. Both can be directly transformed into each other.

Within this paper graphical syntax is shown in examples. The syntax presented here is not complete and preliminary as the final language is still in the standardization process at the time of writing this paper. Nevertheless, the syntax presented here is well suitable to illustrate the usage of the semantic elements and it provides a first glimpse on the envisioned final language.
2.11 Aspect Orientation

SCALE contains constructs that allow aspect oriented programming within the design of compositions as well as on the level of the execution environment with aspects applicable to multiple compositions. We expect AOP to be a powerful tool for enforcing policies and rapid creation of customized service variants through controlled and efficient distribution of new functionality within a service composition environment. One of the goals of SCALE is efficient creation of composite services and of customized variants. This goal is well supported by AOP.

2.12 SCALE feature summary

The following list contains the key properties of the SCALE language

- Technology independent service handling
- Service technology support by means of optional modules
- Late Binding / Loose Coupling
- Constraint Based Dynamic Service Selection
- Constraint Based Control Flow
- Coordinated parallel service execution
- Event and Message Driven
- Aspect Oriented Programming support
- Textual and graphical syntax
- Language subsets targeting different developer groups

3 Reference architecture of a SCALE Engine

A language should stay as much independent of a certain implementation as possible. However, some concepts demand a common understanding of a few high level architectural concepts needed within a suitable implementation of an execution engine.

The objective to stay technology agnostic at language level is addressed by using abstract services in the SOA sense. These services are defined by abstract properties describing for example the function they would provide. These services are also selected based on these abstract properties. This allows the developer of the composition to operate on a high abstraction level. The developer can focus on the overall function to rather than every technological detail.
However, somewhere the abstract service needs to be executed based on the concrete technology of its implementation. This is reached by an architecture consisting of two layers. The composition execution core (CEC) is interpreting the SCALE language, thus operates on abstract services. It is accompanied by a layer of composition execution agents (CEA).

A composition execution agent implements all functionality needed to execute services of a certain technology. For example, a CEA for Web services will be able to receive, evaluate and send SOAP messages. Another CEA might be able to send and receive SIP messages.

Please note that the SCALE language does not define which CEAs are available and how they are implemented. Also the API for attaching CEAs is strictly not part of the composition language as such. However standardizing the CEA API together with the SCALE language makes most sense in order to allow coordinated design of exchangeable CEAs.

All CEAs use the same API towards the composition execution core. It is the main task of a CEA to translate the technology specific details of the external interface into the general concepts of SCALE. For example the reception of an external protocol message might be exposed to the core by means of issuing an event. Is an external message is received, the composition execution engine generates an event and issues this event to the respective composition sessions where then further activities can be performed. These events also bear data received on external interfaces. This data might be mapped into shared state variables of the respective composition sessions when consuming the respective event.

However, the nature and details of the interface between the CEC and CEAs is not subject to this paper and also not part of the SCALE language definition.

Figure 4 A technology agnostic core with modular technology support
The CEA concept encapsulates the details of a service technology into modules, which are in general optional. If the SCALE language shall be used in a service environment that does not contain for example enterprise service busses, the respective ESB CEA in not needed. Furthermore, future technologies can be added without changing the composition core.

A modular and layered architecture as described in this chapter is of course not strictly needed in order to implement an execution engine for the SCALE language. Nevertheless it is extremely helpful to reach an extensible, thus future proof environment. In this paper we assume an composition execution engine implementation following the layers approach discussed in this chapter.

4 Key Elements

4.1 Syntax presented in this paper

In order to illustrate the use of the semantic elements, a graphical syntax is used in this document, which might vary until the final specification of the language is ready. While the semantic elements and a textual syntax will be strictly specified in the SCALE language specification, the graphical representation is a recommendation that might vary depending on the used development environment.

4.2 Skeletons

A skeleton defines a composition of services. The SCALE composition language is used to create a skeleton. The name skeleton is chosen in order to indicate, that the developer only provides a set of abstract placeholders for services and how they relate to each other. The relation is expressed by means of constraints that guide the execution engine which services to select and how to execute them. The developer does not need to know which service implementations are actually available in the execution environment. Also the final execution order is in general not known at design time. Such decisions are left to the execution engine, which fills the abstract skeletons with concrete services and provides an execution structure.

4.3 Actions

Actions are the central element within skeletons in SCALE. An action is used to define any kind of activity to be executed by the engine in order to achieve the wanted composition. A composition in SCALE is to a great extend a set of actions accompanied by constraints that define their relation and content. This chapter provides and overview of the semantic properties of actions and the interpretation of them by the execution engine.
4.3.1 Basic elements and structure

Figure 5 shows four actions. In graphical notation an action is represented by a rectangle. Actions are named and in the example in Figure 5 the action names are X, Y, A and B. Please note, that in this example many properties and elements of actions are still missing and will be successively introduced throughout this chapter.

![Figure 5 A skeleton with four actions](image)

Actions can be nested, thus an action can contain further actions. Action Y for example contains the actions A and B while Action X is atomic.

4.3.2 Service Template – An Atomic Action

One of the most frequently used purposes of actions is being a placeholder for services to be executed. The execution engine dynamically selects a service based on constraints provided by the action.

![Whitelist_Usage](image)

Figure 6 An action specifying a service template

The example in Figure 6 shows an atomic action defining a service template. The keyword “TEMPLATE” is used in order to specify, that this action is a service template, thus the engine is supposed to select a service and start the service when executing this action. In this graphical syntax, the service template is represented by a yellow sprocket.

The example in Figure 6 shows an atomic action defining a service template. The keyword “TEMPLATE” is used in order to specify, that this action is a service template, thus the engine is supposed to select a service and start the service when executing this action. In this graphical syntax, the service template is represented by a yellow sprocket.

After the keyword “TEMPLATE” a constraints expression is provided, which allows defining constraints for service selection. In this example a service is required, which provides user settings as stored in a user profile. Such a service might be described in the service repository with the property “srv”
having the value “user_profile” and the constraint instructs the engine to select a service with this property.

The input parameters of the service can be specified and filled with values. In this example, the action has two input parameters. To the first input parameter the value of the variable userid is assigned, which provides the address of the user. The second input parameter specifies the information to be returned. In this example the string “whitelist_used” is assignend to it. The input parameters are filled with values in the moment when the execution of this particular action / service template is starts.

Also potential output of the executed service can be handled. Here, the action stores the result of the service in the variable whitelist.

4.3.3 Compound Actions

A compound action is an action, which contains nested actions. The action “Whitelist_Preferences” in Figure 7 for example consists of the two actions “Translate_UID” and “Whitelist_Usage”.

Execution of the parent action means execution of the nested actions. Here the nested actions are ordered in a way that they constitute a thread where “Whitelist_Usage” waits for “Translate_UID”. More on the action execution order follows in chapter 4.3.6.

This example also shows the locality of data. All variables are local to an action. At the start of the action, a local copy of the data in the action’s parent environment is made with a snapshot of the values in the moment when the action execution starts. Within an action variables can be changed and new variables can be created, but these changes are by default not forwarded to the actions parent environment. In the example the variable “id” is created within the action in order to store the translated user ID. This variable is not exposed outside of “Whitelist_Preference”.

The only way to forward variable changes to the outside is through the output fields of the action. This field specifies which variables are handed through to the outside. Only those variables specified this way will have an effect on the action’s environment. Thus these output fields are more accurately named “effects” rather than “outputs”. In the example the variable “id” is defined to have an effect to the outside of “Translate_UID” it overwrites or creates the variable “id” within “whitelist” preferences.
In the case of variable “whitelist”, effects are defined at even multiple stages. The effect of “Whitelist_Usage” leads to updating/creating the variable “whitelist” in its parent “Whitelist_Preference”. The effect of “Whitelist_Preference” in turn leads to updating/creating the variable “whitelist” in the outside environment.

Please note that all kind of actions provide the same properties to their environment like for example input fields and effects/output. These exposed properties constitute the API of an action. This means, actions can technically replace each other as long as these action APIs match. The compound action of Figure 7 can for example replace the atomic action of Figure 6.

4.3.4 Actions Summary

The following picture shows the elements of actions. The details of the elements will be explained in the following chapters.
Figure 8  A graphical syntax of actions

Figure 8 shows all possible elements of an action. Some elements like the action name are mandatory, but most elements are optional.

Action Name
Actions have a name that is unique within the actions parent scope. Other parts of the composition skeleton can refer to this action by this name.

Start Handle
The start handle allows to express graphically a starting constraint for the action. The action start is waiting at least until this handle receives a trigger, for example from another action’s finishing handle. Using the start and finishing handles allows to define an static control flow manually.

Finishing Handle
If the action finishes, this handle provides a trigger. This trigger can be used by means of drawing arrows in order to satisfy start constraints of other waiting actions.

Input handle
Input handles are used in order to graphically formulate starting constraints that are based on data dependencies.

Input expressions
Input expressions are used in order to define dedicated input parameters of the action. If the action is for example a service template, these input parameters correspond to the parameters of the service that is about to be started.
Effect variable
All variables are local within an action by default. If a variable shall be updated and written into the parent scope, it has to be defined as an effect. All variables from the action’s local scope that need to be changed also in the parent scope need to be defined as an effect of the action.

Effect handles
An effect handle provides a trigger once the effect variable it is associated with is written. These triggers can for example be used by means of arrows in order to satisfy data dependency related starting constraints of other actions.

Action Expressions
These fields contain expressions that control the action operation. Thus the expressions also determine the type of the action. For example constraints can be given if the action shall do a service selection, or a starting condition can be given by means of an expression. The expressions are optional. An action without any expression is just a basic compound action. The expression usually starts with an keyword determining what the expression is about. For example the keyword TEMPLATE would start an expression that provides constraints for service selection. Multiple lines of expressions are possible, however not all combinations of expressions are meaningful and allowed.

Nested Actions
An action can be compound, which means that it contains other actions. If there are no nested actions the action is atomic. Nested actions are encapsulated, thus not visible from outside its parent.

Action symbol
Actions can have different types depending on what they are used for. They can for example select and invoke services, thus being a service template, or they can be compound. In order to better visualize the functionality, it is possible to assign different symbols to different types of actions. Which symbol is chosen can for example be derived automatically from the keywords used in the action expression lines

Local Declaration Field
This optional text field allows textual declarations and definitions for the local scope of the action. It is used for example to define named scopes, or to formulate event bindings and event filters.

4.3.5 Action expressions
The action expression is a key functional element of an action. It usually starts with one of the following keywords, which also determine what type the action is. The meaning and usage of these different action types will be described in the later chapters of this document.

TEMPLATE
Service template for constraint based selection and execution of services. Templates are atomic actions
SELECT
Constraint based selection of services. Result is a list of services which satisfy the constraints. Atomic Action

INVOKE
Execute a service. Input is a selection of services. Atomic Action

DELEGATE
Delegate the execution to another action

COMMAND
allows explicitly giving control and configuration commands for execution agents

SSM
allows data manipulations like assigning a value to a variable

IF
Conditional execution depending of two alternative sub-skeletons

CASE
Conditional execution with multiple alternatives

STARTIF
is an additional condition that needs to be fulfilled for starting the action. The action start is waiting until also this condition is fulfilled.

USEIF
is an condition for starting an action. While STARTIF defers the start of an action, USEIF determines if the action is considered at all by the execution engine.

FORALL
Instances of the action are created and executed for each element in a certain variable. This variable is usually a list or a map.

FOREACH
The action is executed for each element in a certain variable. This variable is usually a list or a map. The iterations for each of the variable elements are executed subsequently within the same local environment.

WHILE
After an action has finished trigger it over and over again while a condition is true

QUERY
Allows to do database queries. Query type actions will provide the query result through an effect.

EVENTBINDING
Allows the modification of existing event bindings or the definition of a new event binding. This also allows injecting event handlers to global environment.
INCLUDE
Refers to the elements of an externally defined skeleton and places them into
the local environment. These external elements will then be executed as if
they were defined locally.

IMPORT
Refers to an externals defined skeleton and makes its elements available
locally by name. This allows to refer to and use the external elements locally.

4.3.6 Start of Actions – Parallel by Default

In order to execute a composition, actions are executed by the composition
execution engine. Per default actions are executed asynchronously, thus in
parallel. This means that the composition execution engine tries to start all
actions at once with the start of a skeleton. Constraints of many kinds can be
used in order to defer the action start until certain conditions are met, or to
conditionally omit an action altogether. In this chapter rules for starting of
actions are discussed.

If an action has a start handle, at least one instance of it is about to be started.
A bare and unconnected start handle implies that exactly one instance will be
started and the start can be done immediately with the action’s parent. This is
the default starting behavior of actions. Constraints as described below can be
used in order to deviate from this default by imposing further conditions.

Triggers constitute a constraint that changes the starting behavior of actions.
In the graphical notation a trigger is visualized by an arrow. A trigger arrow
pointing to a starting handle implies two things:

1. The action can only be started once a trigger is received and not
   immediately with the parent.
2. Only with a trigger received an instance of the action is created, thus
   becoming waiting for its start.

Please note, that further constraints might need to be satisfied before an
instance of an action actually starts, but each trigger to the start handle
creates a new instance that either starts immediately or starts once all its
starting constraints are fulfilled.

In the example skeleton in Figure 9 A1 has an unconnected start handle and
no further constraints that impose further conditions on the start, thus an
instance of A1 starts immediately with the skeleton start. The same is valid for
A2. Both A1 and A2 can start immediately, thus they are started in parallel.

An instance of A3 is also supposed to start, but a data dependency was
defined. The actual start is deferred until a trigger indicating the update of
variable v1 is received from the effect of A1. Once the variable is updated
within A1 also A3 can start.
The start handle of A4 is connected to the finishing handle of A1. A4 can start once it receives a trigger. Here the trigger will be generated when A1 finishes. This way an order in the execution can be defined explicitly.

ET1 is a trigger sender. The trigger sender shown in this example is an event trigger ET1 connected to A5. A5 therefore waits for the event to happen in order to receive the trigger. Multiple instances of A5 might be started because a new instance is started with every trigger, thus with every event.

In this example A1, A2, A3, A4 and A5 have start handles and the composition engine will create and start at least one instance of them in order to execute the skeleton. A6 does not have a start handle. It is therefore a pure declaration of an action and being purely declarative, the engine will not start it. In this example A4 delegates its execution to A6. This means once A4 is started, actually A6 is used. This mechanism of declaration and delegation allows separating complex actions order to keep the skeleton comprehensible.

Nested actions start with their parent following the same principles as for the overall skeleton. In the example in Figure 10 the nested actions A7 and A8 are started immediately when their parent action B1 is started. Within the local scope of the parent, also declarative actions (A9) and event triggers might be used (ET2 starting A10).
The example in Figure 11 shows how actions might be triggered multiple times. The action A1 is triggered by the event ET1. If ET1 occurs multiple times, multiple instances of the action are triggered and started. Each of these instances has independent local environments.

The start handle of action A3 is connected to the effect of A1. Every time the variable v1 is written by A1 a trigger is issued to A3. This means each time v1 is updated a new instance of A3 is triggered and started. As A3 has a data dependency with v2, all the started instances wait for the variable v2 before they are actually started.

The triggering mechanism and the arrows as graphical elements for conveying triggers are used in order to specify constraints on the control flow of the composition. It is possible to explicitly enforce a certain execution order. Furthermore it is possible to model a data dependency. This is implicitly at run-time causing the engine to execute in a particular order which regards the dependency, thus fulfills the constraint.

4.3.7 Action Environment

The term environment describes in principle the local data of an action. In order to control race conditions all variables are local by default and valid within an action. At the start of an action the variables of the action’s parent is copied in order to create a new local environment. Once a local environment is created for an action, writing to a variable by this action stays local by default. Effects need to be specified in order to explicitly allow an action to write into a variable within its parents environment.
4.3.8 Trigger Sender and Receivers

The graphical elements shown in Figure 12 are trigger senders and trigger receivers. They can be used to specifically issue a trigger based on certain conditions or to consume a trigger. The trigger sender is usually connected to action handles. The trigger receivers consumes triggers.

![Figure 12 Trigger senders and trigger receivers](image)

The generic element trigger sender or trigger receiver is used in different contexts and for different purposes, like for example:

- **Event handling**
  Trigger senders are used to trigger actions in order to do event handling. A trigger receiver is used in order to explicitly generate and issue an event. Event handling is described in Chapter 4.6.

- **Conditional execution**
  Trigger senders can be used together with a condition for execution. A trigger sender would correspond to a certain result of the condition evaluation and trigger those actions, which shall be executed under that condition. Conditional execution is described in detail in Chapter 4.3.11.

4.3.9 Trigger Operations

Actions are started on triggers. If multiple triggers are received at an action's start handle, the action is also started multiple times. In order to have better control over the trigger distribution, elements for trigger consolidation are introduced:

![Figure 13 One-time trigger and accumulation elements](image)
- One time trigger element
  This element will only propagate the first trigger arrived. All subsequent triggers will be discarded. This element corresponds to an Exclusive OR operation applied to trigger reception.

- Accumulation
  This element only propagates a trigger, if all branches where triggers could be received from have actually delivered a trigger. This element corresponds to an AND operation applied to trigger reception.

Note, that there is no distinct element for an OR operation applied to trigger reception, because this is the default behavior.

Figure 14  Trigger operations

Figure 14 shows an example for trigger operations. The action A3 can be triggered multiple times whenever the action A1 finishes, whenever the action A2 finishes and whenever the event ET1 occurs.

A6 only triggers once, after the first reception of a trigger from either A4 or A5 or ET2, whatever came first.

A9 is triggered after at least one trigger is received from A7, A8 and ET1. This means each of them needs to have issued at least one trigger. If each of them has issued two triggers also A9 is triggered two times. So the accumulation element still allows multiple triggers to pass.

Note, that this can not only be used for triggers towards action start handles, but also for triggers used to model data dependencies between actions.

4.3.10 Finishing an Action

An action is finished, if its atomic function is executed or if all nested actions, which were supposed to be executed, are finished. The question is, which actions are supposed to be executed? What actions the parent actions will be waiting for before finishing itself? And, what are the concrete conditions that tell if an action can finish or not?
For an atomic action finishing depends to a great extend on the function performed by this atomic action. The general rule is that the atomic action finishes once it has executed and concluded its function. The function of a service selection would be for example performing satisfaction of selection constraints based on the service database. This function is concluded after the service database has delivered an answer and a resulting list of services is available.

Another example is the finishing of a service invocation. A service invocation action tries to initiate service execution. The details of this invocation depends on the service at hand and therefore on the execution agent used and on what is expected from the service. Do we need to wait for example for a confirmation message or even for a service’s result value being delivered? Thus the finishing of the service invocation action depends implicitly on the technological details of the service.

Please note, that the finishing of an action – even of a service invocation action – is in general decoupled from the service execution. The service might be an independent process and the action only uses an API towards this external process. In this case the action would finish, once this API usage is over.

For a compound action the same general rule applies that it finishes once its inherent function is executed and concluded. For a compound action its inherent function is determined by its nested actions, thus it is possible to define rules for finishing the compound/parent action based on its nested components.

A compound action finishes if NONE of the following conditions is met any more:

- There are instances of nested actions still being executed. Nested actions might have been started and they have not yet finished themselves, thus also their parent does not finish.

- There are instances of nested actions waiting for their actual start. These nested actions have received a trigger to its starting handle, but they cannot consume this trigger, because there are other starting constraints at this action not satisfied yet. This can also be phrased as: A parent action does not finish while in its local scope there are still starting triggers pending.

- There are mandatory but still pending event trigger points. A mandatory event trigger point corresponds to an event, which must occur at least once. Thus, the action in which such a mandatory event trigger point resides must wait. Please note, that this condition can not only change because the event was finally received, but it can become not mandatory any more for example if the mandatory property was tied to a scope.
4.3.11 Conditional Execution

Conditions for alternative execution are a basic element of programming languages. Also SCALE allows specifying conditions for executing optional parts of the composition. Within the action expression a condition can be specified and based on the results of this condition’s evaluation different trigger senders and actions connected to them will be engaged.

**Whitelist Usage**

```
IF: whitelist == 1
```

**Figure 15 Conditional starting of nested actions**

Figure 15 shows a simple conditional statement. The action expression is started with the keyword “IF” followed by the condition. The condition is evaluated when the action starts. Special trigger senders are used in order to trigger different parts of the skeleton depending on the condition. The trigger senders THEN and ELSE are specific to the IF condition. They correspond to the condition being evaluated as true or false.

The action containing an IF action expression is compound and the conditional triggering can be applied to nested actions. Both, the THEN and the ELSE triggers are optional. If for example the condition evaluates to false and there is no ELSE trigger sender the action can finish without doing anything.

In Figure 15 also an nested action called “Logging” is defined without being connected to an trigger sender of the IF condition. This means, its execution does not depend on the condition. It is treated as usual nested action and it is triggered with the start of its parent. Here “Logging” will be started in parallel with either “Whitelist” or “Blacklist”.

Figure 16 Conditional starting of nested actions

An alternative usage of the IF type action expression is shown in Figure 16. The condition can be split and the possible values can be moved to the trigger senders.

Next to an IF statement with two branches it is also possible to use a higher number of conditional branches. An example is shown in Figure 17. The condition is based on evaluating the variable "preference". The conditions are distributed at trigger senders. In this example the conditions distinguish, if the variable 'preferences' contains the strings ‘SMS’, ‘MMS’ or ‘EMAIL’ in order to send a notification to a user in the preferred way.

If none of the conditions is fulfilled, the branch marked with ELSE is triggered. Also here, the action called ‘No’ is always triggered in parallel to the other actions. It is not part of the conditional triggering but depends on its parent’s start. The Action ‘A1’ is triggered from all three possibilities. In this example the message sending services ‘Send_SMS’, ‘Send_MMS’ and ‘Send_EMAIL’ depend on the actual message text to be available. This text is generated by the service ‘Message_selection’. In the example this is modeled by means of a data dependency. At start of the compound action ‘Message_Sending’ the IF condition is evaluated and the respective alternative is triggered. Due to the data dependency the actual start of the triggered message sending service is postponed until the service ‘Message_Selection’ has issued a message text.

The condition expression can also be completely moved to the trigger handlers and it can consist of a complex expression based on multiple variables and complex operators.
4.3.12 Conditional action start (STARTIF and USEIF)

By using action expressions of type STARTIF and USEIF further conditions for the start of an action can be defined.

STARTIF is yet another constraint that postpones the action start. It works similar to the input handles because the action start is waiting for the condition to be fulfilled.

The usage of the STARTIF expression is shown in the example in Figure 18. The service template action ‘Send_Message’ has a data dependency with the action ‘Check_Urgency’ regarding the variable ‘level’. The STARTIF expression adds the constraint that the start is postponed not only until the variable level is updated, but until it has a value greater than 10.
While the STARTIF expression only defers the start of an action until a certain condition is fulfilled, the USEIF expression adds a condition for executing the action at all. If the action is about to be started, the USEIF expression is evaluated. If it is fulfilled, the action actually starts. If this expression is not fulfilled, the action finishes immediately without being executed.

Please note, that the action with a USEIF expression actually waits for all starting conditions to be fulfilled, before the USEIF condition is evaluated. This includes waiting for data dependencies and conditions phrased by means of STARTIF. Please also note that in case of the USEIF condition being false, the action will not trigger its effects.

![Figure 19 Use of USEIF action expression](image)

The example in Figure 19 is similar to the one in Figure 18. The only difference is the use of USEIF rather than STARTIF. In both examples the action ‘Send_Message’ waits for the variable ‘level’ to be updated. When using STARTIF as in Figure 18 the action is waiting until a certain value was written. When using USEIF, at the first update of the variable ‘level’, a decision is taken if the action is executed at all. If the variable value is greater than 10 the action is started and executed. If the variable value is not greater than 10, the action finishes without being executed.

4.3.13 Loops

Loops are created by means of compound actions. The actions content including its nested actions is what is executed at each of the iteration of a loop. Two types of loops are supported in SCALE. WHILE loops iterate the compound action while a condition is true. FOREACH iterates through all elements of a list or map and executes the compound action again for each of these elements.

An example WHILE loop is shown in Figure 20. The nested actions are repeatedly triggered and executed while the condition is fulfilled. Here the loop is done again if the variable still contains the value 0. This example implements a regular polling of an external value until it is not 0 any more. The action ‘Wait’ shall lead to a certain delay until the next loop run, thus the next value polling is done.
Figure 20  An example WHILE loop

The FOREACH action expression creates a loop through all contents of a list or a map. An example is shown in Figure 21. The variable ‘userlist’ contains a list of usernames. The loop will iterate through the entire list by triggering all nested actions again for each of the users. The variable ‘user’ contains the list element for which the iteration is done. In this example a Message shall be sent to all users in the list. Note that a FOREACH loop handles all iterations sequentially one after the other. Please also note that a new iteration is done by triggering the nested actions again. The local environment is not reset for a new iteration.

Figure 21  An example FOREACH loop
Another way of triggering actions for elements of lists or maps is the FORALL statement. FORALL creates a new instance of the action for each element in the list or map. All these instances are triggered in parallel. All these instances will have the same input dependencies and also the same effects. The only difference will be the list element they are started for. As separated instances of the action are created, they also operate with separate environments, thus separated local data.

![Message_Sending](image)

**Figure 22**  An example FORALL loop

FORALL is not a loop in the sense of many programming languages, because it starts all iterations at once and within independent local environments rather than looping through them sequentially.

Figure 22 shows the same example use case like in Figure 21 but here it is done with a FORALL. The difference is that now the message sending for all users in the user list started in parallel rather than sequentially. Each of the parallel instances has its own local environment copied from the parent at action start, except the variable ‘user’. This variable gets a different value from the list of users within every started instance reflecting the user for which the action instance is started.

4.4 Flexible Constraints

Constraints are one of the central elements of the SCALE language. Constraints control the selection of services and they also define dependencies between the services which allow the engine to determine a certain control flow. We can therefore distinguish selection constraints and structural constraints.

Selection constraints were already shown above as central data for service template and service selection actions. In such actions these constraints determine the criteria a service needs to fulfill to be suitable for usage.

Structural constraints were also used already. The arrows, which are the bearer of triggers, are such structural constraints. The triggering concept is
used in order to express dependencies, which are considered to be structural constraints.

Constraints can not only be tied to an action element or to trigger arrows, they can be defined independently and centrally and then referred to by name from multiple actions.

![Diagram of constraints assigned and referenced]

**Figure 23** Constraints assigned and referenced

An example of constraints, which are named and defined independently of actions, is provided in Figure 23. This example also shows how they are assigned to and referenced from actions. C1 and C2 are constraint definitions. C1 defines an additional constraint that is assigned to A1 and A3. The assignment is done graphically by means of an arrow towards the action symbol. This means that the constraint expression of C1 is added to the action expressions of A1 and A3. Here additional constraints for service selection are defined.

C2 also defines additional constraints. C2 is referenced by name from A1 rather than being assigned to it by means of arrows. This is another way to consume separately defined constraints. C2 also contains the variable v1. Thus, the actual evaluation of the constraint depends on a variable part. This implies potential data dependencies and this is the reason, why constraint definitions can have input expressions and input handles. These handles and therefore also the data dependencies defined using it will be inherited by the actions this constraint is assigned to. In this example A1 uses C2 and therefore A1 will inherit the input v1.

The constraint expression in A3 contains the variable constr1, which is taken from A2 through a data dependency. This variable contains an entire variable constraint expression, thus the constraints being used at run-time can be entirely dependent on dynamic data.
Figure 24 Alternative constraints setting

Figure 24 shows the same scenario like Figure 23, but all elements are embedded in actions. No stand-alone constraints are used. Please note that multiple action expression lines of type TEMPLATE will still lead to a single service being selected. All selection constraints will be accumulated in the sense that all need to be satisfied by the selected service. This means, they are implicitly connected by an ‘and’ operation.

4.5 Scopes

Scopes are a concept to define and mark sub-sections within the dynamically generated control flow. This is very helpful for working with events. Event can be filtered or can be directed to alternative event handling based on the information of certain parts of the composition are executed.

There are two ways to define scopes: Set-based and flow-based. At runtime the composition execution engine derives from the scope definitions the momentary state of the scope. It could be either active or passive. A scope is active if execution is currently performed within the scope’s members and passive if none of the actions within the scope is currently being executed.

For a set-based scope definition a scope is considered to be a set of actions. The scope is defined by enumerating its members. The scope is then defined by specifying a number of actions being part of this scope. The scope is active while one of its member actions is being executed. A special case of set-based scopes are actions themselves. An action constitutes by default a
scope, which can be used by referring to the action name. All nested actions are considered to be part of the scope defined under their common parent’s Name.

![Diagram of scope definitions]

Figure 25  Defining scopes by means of action sets

The example skeleton in Figure 25 shows a variety of scope definitions. All the actions from A1 to A8 constitute a scope by default. There is no need to explicitly define a scope, which only consist of a single action. Such a scope can directly be referenced by that action’s name. In case of A5, the scope called A5 contains the respective nested actions A6, A7 and A8.

User defined scopes can be created by providing a Scope definition statement in the local declarations field starting with the keyword “ScopeDef”. Alternatively a scope can be created graphically by putting a named scope frame around the actions which are supposed to be in scope.

In Figure 25 the scope S1 is defined to comprise the actions A1 and A2. This is shown graphically and by its textual definition in the declarations field.

The scope S2 comprises the A1, A3 and A4. This example shows, that a scope can consist of multiple unconnected sets.

The scope S3 is defined based on scope S1 and A5. This means A1, A2 from S1 and A5 with all its nested actions A6, A7 and A8 is included are included.

The scope definitions and scope names are valid within the local environment. A5 has for example its own local scope definition of S4. S4 is not visible or available outside of A5.
At execution time the engine derives from the scope set definitions the momentary status of a scope. If A1 for example starts being executed, this immediately leads to a change of state to active for the scopes S1 and S2 and for the implicit scope of A1. After finishing A1 the scope S1 is still active because with A2 another element of S1 is triggered. If after A1 finishes, the scope S2 is set back to passive depends on the execution status of A4 and A3. If none of them is executing, S2 becomes passive.

Next to set based scopes, also the more dynamic flow-based scopes can be used. Flow based scopes are also named and they explicitly set to active or passive at certain points in the composition’s control flow.

An example for handling flow based scopes is shown Figure 26. For some of the scopes in this example the setting of active and passive is done explicitly. Scope S5 is set to active if a trigger is received at the start handle of action A1. S5 is set to passive if a trigger is issued at the finishing handle of A2. S6 reaches the same as S5 by means of set based scope definition.

Figure 26 Handling scopes by setting their status explicitly
The scopes can have multiple points where they are set to active or passive. S7 is for example set to passive when A3 issues an trigger and it is also set to passive on receiving a variable update trigger on the first input of A5. This also shows that all kinds of triggers can be used for attaching a scope setting.

Scope S8 becomes active on reception of an trigger to the start handle of A4. and with the first update of the variable v2 it becomes passive. This way, S8 it active from the start of A4 until A4 has updated the variable through its effect.

The scope S9 is defined as a combination of S6 and S8. This means it is possible to combine set based scope definition and flow based scope definitions. S9 is active while the actions A1 and A2 are executed and if set explicitly according to S8.

The action A5 has a nested and local scope defined. Such a scope can be exposed through an effect, thus it becomes available in the parent environment. Here the local scope S4 is exposed under the name S10 to the parent environment of A5. S10 is therefore active while A7 and A8 are executed, although A7 and A8 are hidden within A5.

4.6 Events

Event handling is a major concept for the SCALE language. There are multiple possible sources for events, like for example:

- The reception of an incoming protocol message,
- Events explicitly issued from actions
- Activities on the local environments like an update to a shared state variable
- Session related activity like start and end of sessions
- Execution details, like start and finishing of actions
- Errors and exceptions
- Timers
- Join-points for AOP

The interaction of a composition session with external services is relying on events to a great extent. A SIP message or a SOAP message are exposed towards a composition session by means of events.

4.6.1 Basic Event consumption

An event can be directly consumed within a skeleton. An event trigger point is used for this purpose.
The example in Figure 27 shows some possibilities to use events in skeletons and together with actions. There are three event trigger points for the respective events E1, E2 and E3. This event trigger point issues and trigger once the event occurs. This trigger can be consumed as usual by actions. E1 for example is connected to the start handle of the action A1. This means the action A1 is started once E1 occurs. Actually, a new instance of A1 is started each time E1 occurs, because each time a new trigger is issued.

In this example the same event E1 is also used to trigger A2. Thus, once E1 occurs, both triggers are issued. Instances of both, A1 and A2 are started and with A2 also the subsequent actions A3 and A4 might be started.

Please note, that A4 as it is defined in this example is started with its parent (here the skeleton) rather than explicitly by an event. This means that here only one instance of A4 is started regardless of how many events are triggered. Furthermore, A4 has input dependencies. It waits for the variable v1 to be written by A2. This implicitly means that A4 can only start after A2 starts and updates v1. A2 in turn is waiting for the event E1.

Another event E2 is connected to the second input of A4. This expresses that the actual start of A4 is deferred until E2 occurs and issues a trigger to the input point for v2. This kind of event consumption is helpful in case E2 is an event related to an update of the variable v2.

The event E3 shows, that multiple trigger sources can be used towards a single action start handle and that event trigger points can be added to other kinds of trigger sources.

4.6.2 Explicit Event Generation

Events can be issued explicitly from the skeleton elements. This is achieved by linking triggers to events.
Figure 28  Issuing explicit events

The example in Figure 28 shows some of the possibilities to specify explicitly in a skeleton that an event shall be issued. The event E4 is issued once the finishing handle of A1 or A3 issues a trigger. If A2 is finished, this triggers not only A3 through the start handle, but also the event E5 is issued.

If the variable v1 is updated through the effect of A2, also the event E6 is issued. So also for the event generation the data related triggers can be used. This is also shown for E7. Once variable v1 is created or updated the event E7 is issued.

Explicit events are therefore all events designed by the skeleton developer.

4.6.3 Generation of External Events

Next to explicit events generated directly from skeletons and actions, events issued by the composition execution engine are of high importance. External protocol messages will be exposed as events towards a composition session. This allows designing dynamic reaction on external activity. This also means that even the initial external trigger, which caused the composition to start is exposed as event,

The event name of external events is determined by the respective execution agent. Policies of the engine might also be used in order to determine globally which events are issued to composition sessions at all.

4.6.4 Generation of System Events

Activities of the composition execution engine are another source of events. For example errors and exceptions are exposed like events.
4.6.5 Data conveyed in events

Events can have data associated to them. For example the Message parameters of an external protocol message are conveyed within events and exposed to the handling session as event data.

This data is loaded into the local environment of the event handler when the event is consumed.

4.6.6 Event Locality and Propagation

In order to achieve a controlled event handling it is important to define rules, to which actions events are issued first and how they are propagated.

SCALE distinguishes local and non-local events. Local events can clearly be associated to the local environment of a skeleton or action. It is that environment where the event was issued from. Explicit events are a good example of a local event, but also system events can be local, if their root cause is within a local environment. The propagation rules for local and non-local events are different.

Local events clearly originate in the execution of a particular local action, thus are considered to be local to this action’s environment. By default such local events are consumed locally and they are not propagated to this action’s parent or even globally. However, such propagation of the local event can explicitly be initiated. Local events can be propagated to the parent and they can also be issued on global level. Exposing and propagating the event to the parent is an effect of the action and it is also shown as an effect. The event becomes a local event in the parent environment. Issuing the event on global level leads to treating it as non-local as described below.

Non-local events are system events and external events which cannot be assigned to a particular local environment. A policy defined on global skeleton level determines how a particular non-local event shall be treated and propagated through the skeleton. The default is the following handling: Non-local events are propagated through all actions in a certain order that is derived from the nesting of actions. The general rule is that non-local events are propagated from inside (lowest nested level) out (global level).

To understand the propagation principle it is important to understand the tree like nature of the action nesting stack. Action nesting and parallel execution together will lead to multiple nesting stacks in parallel, which are also dynamically changing throughout the composition execution. As a result the nesting of actions has a tree like structure with the global skeleton level as root of the tree and the lowest nested level as leaves. An example is shown in Figure 29. The example shows a skeleton with nested actions and the respective nesting tree.

If there are multiple nested branches, the propagation of non-local events starts independently at all leaves. If a non-local event is issued to this
composition session, the event is first offered to all these actions, which are the endpoints of the nesting tree.

Figure 29  Action Nesting Stack

If a non-local event is issued to an action, which has no event handling defined for it, the event is automatically propagated in the nesting tree branch and issued to the parent.

If a non-local event is issued to an action and the action consumes it. This eventually stops the event propagation within the nesting branch of the action. The event is not automatically propagated unless the action explicitly propagates it to its parent. This means, from an action’s point of view the event handling for local and non-local events is similar in the respect, that propagation to its parent needs to be explicitly initiated. Please note that consuming an event can stop further propagation within one nesting branch, but does not effect event propagation in other branches of the nesting tree.

At the points where two sub-branches join, the event might be received from both sub-branches. This means an action can receive the same event instance several times, because it was propagated from several of its nested actions. It is up to the action to decide if subsequent receptions of the same event instance will also be handled multiple times or ignored.
Event propagation through the entire nesting tree appears to be an enormous and time consuming effort if the nesting tree is big. However, it can be expected, that only a minority of actions define event handlers for a particular event and only these actions need to be taken into account. The actual propagation tree for a particular event is therefore much smaller than the entire nesting tree.

The event propagation policy on global level can deviate from the described default event propagation method. For a certain event it is possible to determine that it is only issued on global level and not propagated through all nested actions. It is also possible to determine that an event is always also issued on global level. This ensures that global event handling can consume it even if the event was already consumed by all nested actions. This policy is valid per skeleton, thus the same external event, if it is issued to several composition sessions might be treated differently from session to session.

Join point events are part of the AOP sub-system. They are treated differently as they are consumed and handed by the weaving system. More on this can be found in the chapter about AOP.

4.6.7 Event handlers

Event handlers are dynamically composed out of actions. This means, an event is handled by issuing a starting trigger to the starting handle of actions. This triggers action execution and can cause a number of subsequent actions and nested actions to also start. Which actions are started in order to execute the event handler is decided by constraints as usual. The event handler is therefore also composed dynamically by the composition engine. Therefore there is usually no clearly unique and distinct entity, which can be called to be the event handler for a particular event. A number of actions being executed based on a trigger issued, because of an event was received, constitutes the handler.

The event itself is consumed in the skeleton by means of triggers issued through event trigger points. If the event occurs several times, for each event a new trigger is issued, which in turn will trigger a new instance of the connected action. Therefore, with each event also a new composition of an event handler is started.

4.6.8 Event Filtering / Event Binding

Event binding is a very powerful way to consume and filter events before they are further consumed and handled by means of for example trigger senders. This chapter outlines some of the possibilities.
An event binding is a filter for events. The binding is invoked by the reception of an event. It contains an expression, which formulates a condition and as result it allows specifying what shall be done. Figure 30 shows the principle.

Event bindings are defined within the action’s local declaration field, thus the event binding definitions are local. Event bindings are named and can therefore be referenced for applying changes dynamically.

An event binding can map input events of any type event to an explicit output event within the local environment. This mapping can be conditional, thus the output event is only thrown if next to the reception of the input event also a condition is met. Scopes can be used here. An output event is for example only issued if a certain scope is active.

Figure 30  Event Binding

Figure 31  Events and Bindings
Figure 31 shows some possibilities. The event EVENT1 is consumed directly as trigger for the action A1. Additionally through event binding EB1, EVENT1 is mapped to ET1 and ET2. The event binding EB1 issues ET1 and ET2 each time EVENT1 occurs.

The event EVENT3 is issued if actions A2 or A3 finish. The event binding EB2 issues ET2 if EVENT3 occurs while scope S1 is active.

Event handling is by default tied to the local environment. The event EVENT1 within the compound action A5 is for example a different event than EVENT1 in the parent environment.

Within A5 there are some examples of events related to data input and effects. EVENT3 within A5 is for example issued each time a trigger is received related to the input of variable v1.

EVENT2 within A5 is referring to the event of the same name in the parent environment. This means each time EVENT2 occurs in the parent, it also occurs within A5.

EVENT1 within A5 is triggering the effect related to variable v2. While the default behavior of the effect is to trigger each time the variable is updated, this effect only triggers if EVENT1 occurs. This can be used for controlling the effects explicitly.

The second effect of A5 is triggered in case of ET2 occurs. This effect issues EVENT4 into the parent environment. This means, that the occurrence of ET2 within A5 is exposed as EVENT4 to the parent.

### 4.6.9 Dynamic Event Binding

Event bindings can be named and being redefined or removed or added dynamically. This allows dynamic handling of events and it provides a lot of flexibility. The action expression of type EVENTBINDING is used for setting and modifying the event handling.

### 4.6.10 Mandatory or Optional

An event is by default optional. This means the action or skeleton does not wait for it to be received. But in some cases it is essential, that a certain event is received in order to proceed with the composition. For this reason an event can be declared being mandatory. This means action of skeleton does not finish until all included mandatory events were received at least once.
Figure 32  Mandatory Event Trigger

Figure 32 shows an event trigger sender, which is mandatory. Next to this graphical element also the textual event binding definition can be used in order to declare an event as mandatory. This can be reset dynamically at run-time by means of an EVENTBINDING type action. Thus the status of an event being optional or mandatory can change, which effects the action’s waiting for it.

An event being mandatory is required to occur at least once within the instance of its local environment. Further occurrences of this event are optional, but this can be reset dynamically. This means a subsequent occurrence is mandatory again, thus the surrounding action or skeleton will again wait for it to occur.

4.7 Utilizing external elements and definitions

This chapter described two methods of utilizing the elements of a separately defined skeleton. It is possible to include or to import elements. Both, the import and the include statement can be used in the local declaration field of an action or of the skeleton, or they constitute an own action type.

Include: Include all actions and constraints and other definitions of an external skeleton as if it would be directly defined in the current skeleton. From execution point of view there will be no difference if the resulting set of skeleton elements is defined in one skeleton or separated into multiple.

Import: Like include, import refers to an external skeleton being a source for elements like actions or constraints or other definitions and declarations. But unlike including them, import just makes them available by name. To use them, it is necessary to explicitly refer to them. All the definitions and declarations of the external skeleton can be used, but they need to referred to explicitly. For example, the external skeleton might define an action A1 and the execution can be deferred the externally defined A1 action by means of an action of type 'delegate'. The environment for executing the external action would be the local environment from where it is referenced.
5 Data Handling

5.1 Shared State

Within a composition environment, data plays a central role in the inter-work of constituent services with each other and of the composition as a whole with external users. SCALE uses variables for two main purposes. First of all the variables keep all data from the communication of the composition with external users and services. This refers to all incoming data being stored into local variables and in all outgoing data taken from the local variables. For example the values of parameters conveyed in protocol messages will be exposed to the composition session as data in local variables and the input for services being invoked from the composition session is taken from this same pool of data.

Furthermore, it is frequently necessary for the composition to define a state machine. Especially asynchronous communication does not allow stateless participants. In SCALE variables can be used to store state information in order to build state machines.

![Figure 33 The shared state as central data pool for a composition session](image)

The entire set of data of a composition session is called “shared state”. This reflects the usage of the data to be shared between constituent services for coordination and as data pool for all inputs and outputs. The concept is shown in Figure 33. Please note that data is also shared between services and protocols, which belong to different service technologies. For example the user address as received on SIP can be used as input parameter for a web service. Of course the data format might not match. In this case additional translation services need to be added to the composition.

5.2 Locality

For a language and execution environment that is designed to support many parallel and independent services working together in order to implement a composed service, the control of race conditions is essential. Asynchronous operations that are overwriting the same variables are a frequent cause for errors that are also hard to find by testing. The locality of data is a central concept of SCALE targeting this challenge. The strategy of SCALE is to first of all minimize the number of variables, where concurrent access is possible.
whole not restricting the parallel execution of constituent services. Furthermore the SCALE allows controlling the remaining dependencies.

An action has local data inherited from its parent. This means in particular that the data from the parent environment is copied into the local data of the action. Effects of the action are the only way to push values back to the parent. This already minimizes the number of possible conflicts and side effects from parallel execution to those variables, which are specified as action effects. This means SCALE does not avoid race conditions as such, but situations where they might appear need to be explicitly designed by the composition designer by specifying effects. Furthermore, effects have only local influence as they only reach the variables in the parent environment.

Another aspect in controlling race conditions is the possibility to use data constraints as shown in the chapters above. This allows to tie the data used as input for an action to the output of a particular other action. If a variable is written from many sources like other actions or events, the wanted input is still taken by waiting for a particular write from a particular source of the data.

As a summary, data race conditions within the shared state can be controlled by means of the following concepts in SCALE:

- Local encapsulation of data copies within nested actions. This separates the data from parallel executed actions.

- Only effects allow the definition of selective writing into parent data. This limits the possible race conditions to a small set of variables that are in the focus of the developer.

- Data constraints allow the explicit selection of the source of input data for an action. This is a tool to control the remaining race conditions of the variables exposed by effects.

5.3 Annotations

Annotations are optional properties of data elements like for example a variable or the input and effect slots of actions.

Annotations are specified in a key-value-pair fashion. The key is the type of property, and the value is the actual property of the data element.

Annotations are multi-dimensional. This means several properties of the data can be assigned independently to a single data element. This also means that the annotation is a list of independent key value pairs.

A simple syntax for a variable with assigned annotations is:

```
var1[TYPE=string; FORMAT=email_address] = alice@net.com
```

This previous example shows a two dimensional annotation consisting of a type definition and a property describing the format. So the email address of
Alice is stored in variable var1. The annotation is used to define, that var1 contains a string and that the string has the format of an email address.

When executing an operation with the variable, the way the annotations are handled is not specified by SCALE. SCALE only allows annotating data elements. The functionality using annotations is an externally defined and optional. This means that functionality to set, remove or otherwise manipulate the annotations is within an optional external component. The core engine will ignore them. The only exception to this is data-types as described in the following chapter.

5.4 Data Type System

In SCALE variables can be used without explicit typing and without declarations. Variables are created on demand and the engine selects the type which is fitting best to the data. The engine also performs data conversions if the variable is used in a way that demands a certain type. For example, the engine automatically tries to convert a value into a numeric value once this variable is used within an arithmetic operation.

This way of handling data-types is very intuitive in most of the cases. However, SCALE allows using explicit typing by means of annotations. The data-type annotation is identified by the keyword "TYPE". It is the only annotation which is by default recognized and handled by the composition engine. Nevertheless it is still optional to be used.

Basic data-types supported by default are String, Integer, Real and Boolean. Additionally and also by default the complex data-types List and Map are available. A list is a set of variables referenced by an integer index. A Map is a set of variables referenced by an key value. A Map is therefore similar to a hash table.

5.5 Optional External Type Extensions

Registration of new types and respective handlers for type related operations is possible through a dedicated API of the SCALE engine. With a new type also the implementation of certain type conversion routines and operators needs to be registered.

A new type might be needed in order to define data handling that is specific to a service technology. This is in particular the case if complex data constructs are needed. Their details might be lost when storing them in variables only relying on the basic data types of the engine. As these complex types are specific to the service technology, they are defined external of the core engine and together with the CEAs.

One example where external types might be needed is the storage of a SIP message. It is structured in parameters, which in turn can also have a complex structure.
Another example is SOAP based Web Services. Their data exchange relies on XML and thus can have a complex structure.

Both examples show technology dependent specific data that shall therefore not be integrated into the core SCALE engine. Having additional types defined in optional external modules together with CEAs means that the engine stays future proof in two respects: In case new service technologies will evolve, their specific data handling can be added in a modular way. Furthermore no data type specific to an old service technology stay in the core engine once this technology is removed thus leaving the system clean of legacy.

The skeleton header (declaration field) optionally contains dependencies to certain extensions and their minimum version number. Depending on the execution engine setup, the dependencies could be verified when a skeleton is loaded. Missing optional components could be detected.

6 Heterogeneous Composition Examples

It is an intrinsic and basic principle of the composition approach behind SCALE that the technological details of a service are only known after the abstract selection and with the binding and service invocation. The selection of services as such is in general agnostic of the technology. Only after a service was selected the composer deals with the technological details of the service and invokes it accordingly. It is important to note that technology specific invocation might imply a fundamentally different activity to perform in order to cope with potentially very different service usage roles and topologies. The best example is the fundamental difference of SIP services and Web Services as described above. Thus, for each service technology the meaning of “invocation” might be different and so is the further control of the service execution.

So far we have identified three fundamental ways of consuming a service as part of a composition:

- Request-response:
  A request is sent to the service and the user waits until a result is delivered.

- Asynchronous message exchange:
  A service is a process which is executed asynchronously to the client process. Communication between the processes by messages.

- Service Chaining:
  The service is an self contained entity allocated on a message chain that might be part of an end to end session.

This chapter shows how to deal with these service usage cases in SCALE composition.
6.1 Request-Response service usage

Service consumption with a request-response model is comparably simple because the interaction between the user and service is based on a single request being sent to the service and a single answer being returned. Usually this is performed by the user waiting for the response. The consumption of Web Services usually follows this interaction model.

![Diagram of service usage](image)

**Figure 34** Weather information – a number of basic web services

Figure 34 shows an example of a couple of a simple weather information service based on Web Services that follow an request-response usage mode. The action `Get_location_for_user` selects and invokes a service that will provide the geographical location of the user. As the selected service is a web-service being consumed in request-response mode and synchronously, the action finishes after the location was received in the service’s response.

The other service template type actions of this example work similarly. For example the action `Get_user_reply_preferences` accesses the user profile in order to determine the preferred way to send an answer to the user. `Get_weather_for_location` provides an weather forecast for a specific geographical location. The compound action `Send_preferred_message` is waiting for the message preference information and the text to be sent, before it decides if the sending of an email or the sending of an short message will be triggered.

6.2 Persistent Processes

If the composition needs to communicate with another independent process, this communication will usually consist of multiple incoming and outgoing messages that are asynchronous. The sending of messages will be done
through service template type actions. The message sending is therefore modeled as service invocation. However, this service invocation does not necessarily need to wait for a response.

Responses will be asynchronously received messages. They are exposed towards the SCALE execution session as events. In order to coordinate, a state-machine might be needed, which can be defined using variables.

![Diagram](image)

**Figure 35** Stock Exchange Monitoring and order generation – coordinated communication with persistent service.

The example in Figure 35 implements a composition that subscribes to a stock exchange information system in order to get information of a particular share. It is assumed that there is a service, which can regularly send updates of the share price. This example subscribes to this service with the share the user is interested in.

### 6.3 Composition of SIP services

Although the roles and topology in using SIP services differs significantly from e.g. web services, SIP services are selected based on constraints and actions. From the composers point of view it is first of all important to take the decision and select which service to put on the chain. This task is mainly guided by selection actions and their selection constraints.

Once being on the chain, the service is a self contained unit taking his own decisions based on the end-to-end signaling it is listening to. Crucial in such a topology is the order in which service are selected in order to build the service chain. The second decision of composer is therefore where the service shall
be allocated. In practice, within the dynamic process of chain building this “Where” decision regarding the service order is actually a “When” decision. The chain is built successively one service after another. Thus, this part of the composer decision is subject to constraints, which are guiding timing and skeleton execution order.

Two or more services without any direct or indirect dependency regarding their execution order can be specified in SCALE. They are considered to be independent of each other. If these services are instantiated with SIP services, no clear order for these services to be allocated in the SIP service chain can be derived from the skeleton in order to guide the composer. However, SIP service invocations need to be serialized in order to create an ordered chain. The composer will put SIP services onto the service chain in the order in which they are selected. If there are no ordering constraints, any resulting order, thus any resulting service sequence on the chain is possible. This means, if the order in which the services reside on the chain matters, additional constraints are needed to express this dependency. It is often the case that the order of services is crucial in SIP. The same services put into a different sequence will usually result in a significantly different user experience.

![Diagram](image)

**Figure 36**  skeleton resulting in SIP service chains

The example shown in Figure 36 provides an composition consisting of 3 services. Here explicit ordering is used in order to express, that the execution
order shall be Service 1 before Service 2 before Service 3. The execution is triggered by the event sip_invite, which is bound to the reception of the SIP INVITE method from IMS. INVITE is a typical first forward message in order to establish a communication session with a user.

In this example the skeleton is used in order to compose three services at reception of an invite. Here, for the sake of this example, all three services will be instantiated with SIP services. A heterogeneous example follows below. Figure 37 shows the example communication between the users, the IMS system, the composition engine, and the three SIP services at establishment of the end-to-end SIP session with the two users BOB and ALICE as the endpoints.

On reception of the INVITE message from BOB, IMS contacts the composition engine in order to retrieve services to be put into the end-to-end session. The composition engine will for example start the skeleton shown in Figure 36.

According to the skeleton, the composition execution engine will start building the composition by selecting and starting service 1. As a SIP service was selected, starting it means telling IMS which service to put onto the session chain. IMS will send the SIP INVITE to the selected service 1, where an instance is started. Service 1 returns the SIP invite to IMS and is now logically allocated in the chain. IMS will then ask the composition engine, if there is more to compose. The engine will then proceed similarly with services 2 and 3.

Please note, that from point of view of the composition engine, the service template action for service 1 is not finished before the AS reports about the status of the SIP service execution. Usually this coincides with asking the
composer for the next service to add. This asking for more services might implicitly confirm that the previous task of adding a service was done.

The service chain resulting from this composition is shown in Figure 38. After all three services were selected a command is given, which will result in indicating to IMS that there are no more services to be put on the chain and IMS will route the SIP invite to the destination in order to conclude the end-to-end session setup.

![Figure 38 The resulting SIP service chain](image)

Please note that the skeleton does not necessarily finish after instructing IMS to proceed with SIP message processing. However, no more SIP services can be selected, because the SIP chain building was finished and dynamic additions and changes to an already established SIP service chain are not allowed according to IMS/SIP standards. The skeleton developer has to ensure compliant behavior. Nevertheless other types of services can be used. It would also be possible to capture further SIP messages within this skeleton, thus within the same composition session.

### 6.4 Composition and control of SIP Service Chains

After the initial forward chain setup of the end-to-end communication session, messages can be propagated over the SIP chain both in forward and backward direction. Usually the services, which reside on the chain listen to the messages when they pass by. The services can autonomously decide if they ignore a message or if they become active by for example issuing an message themselves or by altering the original message’s content before forwarding it.

The composition developer might also want to participate in and have control over the signaling on the SIP session. This means, the composition engine needs to be allocated also on the SIP chain being able to catch and evaluate signaling. The reception of a message from SIP can be modeled as event reception in the composition session. Just getting an event about the message reception is not enough, because it is very important to distinguish where on the end-to-end signaling chain the message was caught. In order to control
which events will be generated agents can be configured at the initial building of the chain to stay on the chains like services and to generate certain events, which can then be distinguished.

![Diagram showing SIP service chains with agents for subsequent message handling](image)

**Figure 39** skeleton resulting in SIP service chains with agents for subsequent message handling

The skeleton in Figure 39 shows a complete use case with the SIP services Service1, Service2 and Service3 being put onto the SIP chain. Additionally two agent instances are allocated on the SIP chain at different locations. These agents are configured to throw the events E1 or E2 if a message is received at their respective location. Two event handler bindings are specified in order to filter out SIP messages of type 486 BUSY and to bind the events to two different handlers. Although the message to be caught is the same, different event handlers are triggered depending on the location on the SIP chain where the message did appear.

The skeleton can do a couple of actions. All a service that resides on the chain is allowed to do, can in principle also be done by the composition engine. This includes for example evaluating and modifying the received message before it is propagated. The modifications will be fed back into the SIP chain at the location where the agent did break out with an event. It might also be possible to initiate a partial release of the SIP chain and start routing again towards a
new destination with new services. The decision to do so, the control over the exact point where to break the chain and the composition of the new leg can be controlled in detail by means of skeleton elements.

![Figure 40 The resulting SIP service chain](image)

Figure 40 shows the SIP service chain that corresponds to the example skeleton from Figure 39.

Although the use of event generating agents was motivated by and demonstrating for SIP service chains, this mechanism is a generic one. It relies on basic elements of the SCALE language like events, event handler bindings and actions to provide commands to the service execution agents. These techniques can therefore also be used together with other service technologies. They are not bound to SIP/IMS.

### 6.5 Heterogeneous composition of SIP and Web Services

The examples shown above have demonstrated how to use the SCALE language in order to generate composition for a couple of service technologies with different modes of service usage, taking into account their special characteristics. A heterogeneous composition would go one step ahead and allow composing services from various technologies within a single composition. SCALE is designed to allow exactly this.

At its core constraint based service selection is focused on the provided function and agnostic to the technology which implements that desired function. The technological details will be taken into account only at service invocation rather than at service selection.

A simple heterogeneous example that combines web services and SIP services is shown in Figure 41. On an SIP invite first an action is triggered that retrieves the user preference with respect to blacklist or whitelist services to be used for the SIP sessions. This service can for example be instantiated with a web service that retrieves this information from a user database. As a result, the preference of the user is stored in a variable. The action Black_or_whitelist evaluates the user preference and executes the respectively selected service. As these are SIP services, execution means that the respective service is linked into the SIP chain.
6.6 Practical considerations of technology hiding

Should we really completely hide the underlying service technology? Although the high level and data driven service selection mechanism and SCALE in general would allow this, there are a couple of problems once the comparably simple request-response style of service consumption is left an the constituent services communicate with other services and the composer in an more complex way.

Experience with SIP has shown that composition is for some service technologies not just a selection and execution of a set constituent elements. Control over the topology of the service usage is crucial. It is therefore hard to exchange services, where such topological knowledge is important with services, where this is not relevant due to its different technology and implementation. Even if the two services will in the end provide a similar service to the end-user, the way they reach this and the way the service interacts with other services and the composer differs significantly and this reflects back into composition skeletons. Therefore, in some cases a skeleton is practically best written having a certain technology in mind at least for some of the used constituent components.

Another observation is that data and service parameters are often available within a certain format. The format at hand regularly depends on the technological world of the used service. Although the data can be semantically compatible by presenting the same data just a little differently, conversion is needed. If the technology of a service component varies, so might the format of the data and the needed conversation. In SCALE we have introduced optional data annotations. They can be used to implement a system that automatically detects data type and format mismatches and dynamically adds a correct translation service. Such a data handling and translation system would be an optional add-on. For the time being, actions containing data
translation services need to be added to the composition skeleton by the skeleton designer.

In the end the question is, do we really need to reach a strict abstraction of the underlying technology? Practical experience has shown, that the composer and therefore also its intrinsic language are tools for integration. It is often the central property to have means at hand which allows using multiple service technologies together in an abstract way and to resolve service interaction problems with powerful and high level tooling. In this respect the SCALE language provides a well balanced approach. It allows a high abstraction level in order to stipulate efficient development of compositions. On the other hand SCALE also allows exposing technological specialties where necessary and where not having them exposed and controllable is more a burden than strict technological independence would be helpful.

7 Aspect Oriented Programming in SCALE

The possibility to use aspect oriented programming is an inherent property of SCALE. A detailed description of the AOP framework exceeds the scope of this paper. However there are a few guiding principles, which outline the idea:

The SCALE AOP framework is using a join-point model, which is based on core semantic element of the language like actions, constraints, events and scopes. These elements constitute join-points and advice can be injected at these locations. It will be possible to execute advice before, after or around an skeleton element. Around means in this respect, that the advice replaces the original element.

The AOP framework is based on online waving. This means that the advice code, thus the aspect is linked into the skeleton just-in-time while execution. This means, the conditions for using a particular advice can therefore be phrased based on the runtime data. Join points generate events once their base element is executed/used. The evaluation of weaving instructions and therefore the point-cuts is triggered by these join-point events.

The advice is itself implemented using SCALE language constructs. When advice is executed it is executed within the local environment of the join-point.

Join point events are always propagated from local scope of the generating element up to global execution environment level (even above skeleton level)

Weaving instructions, and therefore aspects, can be assigned locally or even globally. The advice action need to be available, thus declared on the same level as the weaving instruction.

Global aspects are skeletons which contain weaving instructions (point-cuts) and which refer to advice implementation. Registering such an aspect Skeleton in an composition execution environment would make this aspect by default applicable to all skeletons being executed within this environment (also applicable to other aspects). This means AOP can be used within an skeleton
or it can be used on execution environment level, thus being applicable to all executed compositions. The first possibility is an additional tool for the skeleton designer in order to reach increased modularity. The latter possibility is well suited for global enforcement of policies and for central monitoring and control of the composition activities.

8 SCALE at Simple and Expert level

This document describes many advanced features of SCALE. Some of them provide very detailed and advanced means to control the constraint based composition process. The full scope of SCALE is not always necessary. It is therefore planned to define simple sub-sets that are easier to use. This targets less experienced users and enables them to do simple compositions without the danger to get lost in the advanced features. However the distinction between a simplified language version and the full language is beyond the scope of this paper and will be outlined in the language specification.

9 Conclusion and Outlook

In this paper we have outlined the general ideas and concepts envisioned for a new composition language to enable composition in heterogeneous service landscapes. The target is open standardization at relevant and widely accepted organizations like OASIS. It is also a clear intent of merging this language with future versions of BPEL in order to avoid a scattering of the composition technology landscape into specialized and domain specific approaches. However this is a long process.

The next steps will be the first draft of a language specification containing a complete specification of the semantic elements and also a proposal for a textual syntax. We also plan to start a reference implementation of a composition execution engine and related tooling based on SCALE.

For the time being this language is a work in progress. Comments and suggestions are very welcome and highly appreciated. We want to reach a language that has added value to the services community, thus the community is invited to join in.

10 Glossary

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3GPP</td>
<td>3rd Generation Partnership Project</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect Oriented Programming</td>
</tr>
<tr>
<td>AJAX</td>
<td>Asynchronous JavaScript and XML</td>
</tr>
<tr>
<td>BPEL</td>
<td>Business Process Execution Language</td>
</tr>
<tr>
<td>CAMEL</td>
<td>Customized Applications for Mobile networks Enhanced Logic</td>
</tr>
<tr>
<td>CEA</td>
<td>Composition Execution Agent</td>
</tr>
</tbody>
</table>
CEC  Composition Execution Core
ESB  Enterprise Service Bus
IMS  IP Multimedia Sub-System
IN  Intellignet Network
OASIS  Organization for the Advancement of Structured Information Standards
SIP  Session Initiation Protocol
SOA  Service Oriented Architecture
SOAP  Simple Object Access Protocol
XML  Extensible markup language

11 References


