Modeling a Distributed Intrusion Detection System Using Collaborative Building Blocks

Linda Ariani Gunawan[1], Michael Vogel[2], Frank Alexander Kraemer[1], Sebastian Schmerl[2], Vidar Slåtten[1], Peter Herrmann[1] and Hartmut König[2]

[1] Department of Telematics
Norwegian University of Science and Technology (NTNU), Trondheim, Norway
e-mail: {gunawan, kraemer, vidarsl, herrmann}@item.ntnu.no

[2] Computer Science Department
Brandenburg University of Technology (BTU), Cottbus, Germany
e-mail: {mv, sbs, koenig}@informatik.tu-cottbus.de

Abstract

Developing complex distributed systems is a non-trivial task. It is even more difficult when the systems need to dynamically reconfigure the distributed functionalities or tasks. Not only do we need to deal with the application-specific functionalities that are intricate, but we also have to handle the complex logic of coordinating the distribution and relocation of tasks. In this paper, we model an intrusion detection system that distributes its analysis units to a number of hosts and assigns fine-grained analysis tasks to these hosts in order to cope with the rapid increase of audit data from today's IT systems. To avoid the overloading problem, this system also needs to manage the transfer of some tasks from a host to another one. To develop this complex system, we apply the model-based engineering method SPACE. In particular, we show that the collaborative specification style of the method can significantly reduce the development effort. Also, the formal semantics of SPACE ensures the correctness of important design properties.

1 Introduction

The engineering of a complex distributed system is constituted as one of the most complex construction tasks for developers [Jen01]. Such a system consists of a large number of interdependent parts that must interact to realize certain functionalities [RWMB98]. To guarantee a correct execution of a system, it has to be designed and analyzed carefully. The development is even more difficult for systems that reconfigure the distributed functionalities dynamically. Here, we also need to deal with the complex logic of coordinating the relocation of distributed functions.

A distributed Intrusion Detection System (IDS) is an example of a complex system that distributes its analysis capability and dynamically reconfigures it to adapt to changing contexts. Due to the growing transmission and computing performance of networks and end nodes, IDSs have to cope with the rapid increase of the audit data volume (see [CDK+09]). One solution is to distribute the analysis functions among a number of hosts. But, the workload of the analysis components may vary because of the continuous changes in CPU load and in network traffic. Therefore, the IDS should enable a distributed analysis in which the analyzer tasks may be freely moved among different hosts according to the current network and processor loads.

When realizing such a distributed IDS, one faces two kinds of challenges with regard to complexity: the computational complexity of the pattern recognition in audit data streams and the coordination complexity needed to distribute and relocate the analysis task. Since these complexities are of different nature, they require a different handling. The logic to recognize attack patterns is formulated in the form of algorithms in a programming language that are implemented and executed locally to a component. We refer to [MSK05] for details on efficient attack pattern recognition.

The logic to manage the distribution and relocation of the IDS functions, on the other hand, is highly interactive in nature. Such coordination requires the collaboration of several components [CPV97]. To facilitate the development of such a complex system, we use model-based engineering. In particular, we adopt SPACE [Kra08] and its tool suite Arctis [KBH09] in which collaborative behavior can be explicitly specified in the form of UML 2.x activities and collaborations. This collaborative specification style enables a higher degree of reuse than component-based models [KH09]. The semantics of these diagrams is formally defined [KH10] which makes it possible to prove important correctness properties of a specification. The models are self-contained and can be automatically transformed into executable code.

The focus of this paper is to show that SPACE and Arctis are not only suited to develop more traditional distributed services which we already could prove [Kra08], but are also helpful to create systems comprising dynamic reconfiguration, i.e., the assignment and transfer of analysis functions to different hosts in a distributed IDS. We first model the IDS core functionalities executed in a single analysis host and discuss thereafter how to specify the distribution by rearranging its building blocks and by adding further Arctis blocks providing communication and hand over of analysis functions between various hosts. In Sect. 2 we introduce some fundamentals on IDS and signature modeling needed for the

further understanding of the paper. The use of SPACE to model the IDS core functionality is described in Sect. 3, while Sect. 4 presents the distribution process. In Sect. 5, we discuss the incremental verification of the system and estimate the reduction of development effort by using our specification technique. Related work and concluding remarks are presented in Sect. 6 and Sect. 7, respectively.

2 Signature-based Intrusion Detection

Intrusion detection systems consist of sensors and analysis units. *Sensors* monitor the activities of IT systems to be protected. The behavior of applications and the operating system on observed hosts are recorded. IP packets at switches or routers can also be logged as well. From these observations, security relevant activities are extracted and sent as audit events to an *analysis unit*. IDSs apply either pattern anomaly or misuse detection. Anomaly detection scans the current system behavior for deviation from pre-defined normal behavior. Misuse detection searches for traces of security violations in the audit data trail using known attack patterns – the signatures. Misuse detection is applied by the majority of IDSs used in practice. The implementation and configuration of misuse detection systems are simpler and enable a significantly higher detection accuracy compared to anomaly detection. Therefore, in the following we focus on misuse detection based on signature analysis. We model signatures in an Event Description Language (EDL) which uses a Petri net like description principle [MSK05].

2.1 EDL Signatures

EDL descriptions consist of places and transitions connected by directed edges. *Places* represent relevant system states of an attack. Hence, they characterize the attack progress. *Transitions* describe state changes triggered by audit events which are parts of the audit data stream recorded during an attack. An example of an EDL signature with four places (P_1 – P_4) and three transitions (T_1 – T_3) is depicted in Fig. 1.

Ongoing attacks are represented in EDL descriptions by tokens on places. Tokens can be labeled with values like in colored Petri nets. A place defines zero or more features which specify the properties of tokens located on this place. EDL distinguishes four place-types: initial, interior, escape

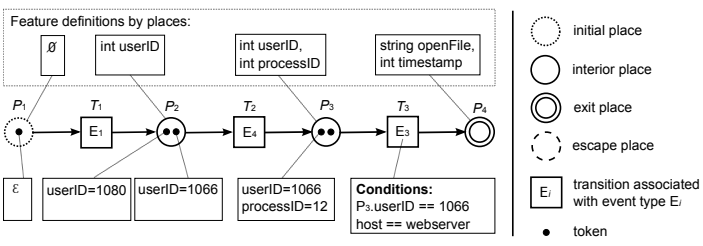

Figure 1: EDL Signature Example

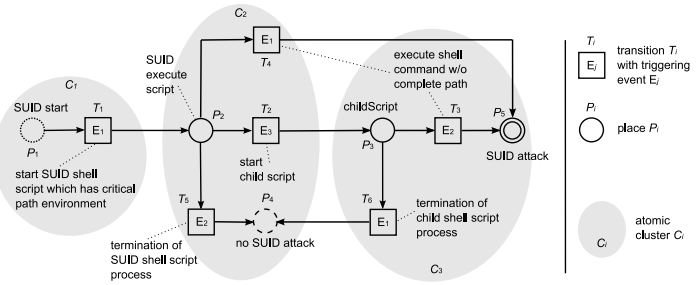

Figure 2: SUID Signature Example

and exit. *Initial places* are signature's starting places which are marked with an initial token at the start of analysis. The *exit place* represents the completion of an attack instance. Thus, a token reaches this place implies that an attack has been detected. *Escape places* indicate the occurrence of events that make the completion of an attack instance is impossible. Therefore, tokens reaching this place-type are discarded. All other places are *interior places*.

A transition that is triggered by an event of a certain type can also contain a set of conditions. As shown in Fig. 1, these conditions can specify constraints over certain features of the triggering event (e.g., `host=webserver`) and token values (e.g., `P3.userID=1066`). The evaluation of these transition conditions requires CPU time depending on the complexity of the conditions and the frequency of the evaluation, which is determined by the number of occurring events in the audit data and the number of tokens on input places of a transition.

2.2 Distributed Analysis

Today's IDSs face a rapidly growing amount of audit data which often forces them to drop parts of these data. To cope with this problem, the required analysis effort can be distributed among a set of cooperating analysis units on different hosts. For this, the signature base of the IDS can be simply split up into parts for the distinct analysis units. But, this may lead to a radical increase of communication effort due to the necessary duplication of audit data for each analysis unit. Therefore, a more sophisticated signature distribution is required. To achieve this, we identify minimal tasks (*atomic clusters*) in a signature, which can be independently assigned to different analysis units. Splitting up signatures into minimal tasks also enables the optimization of the necessary communication effort required to distribute and duplicate the logged audit data for the distributed analysis units. By pooling atomic clusters which analyze the same type of audit events and assign them to the same analysis unit, events of this type only have to be sent to the subset of the analysis units as necessary.

As an example, we consider an EDL signature that describes a typical multi-step attack on a Solaris system in Fig. 2. The gray shaded spheres show the three atomic clusters, this signature can be partitioned into. If atomic clusters are assigned to different analysis units, a token forward

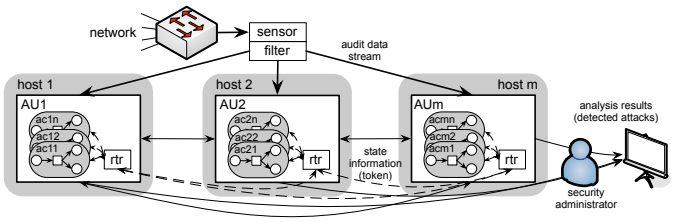

Figure 3: Distributed IDS Overview

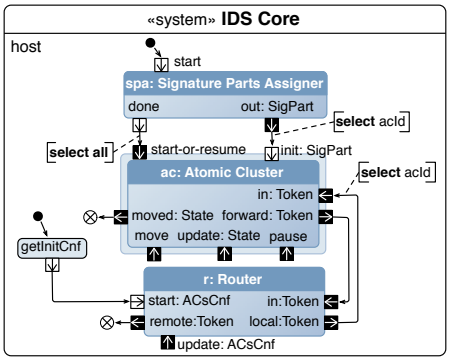

Figure 4: IDS Core Functionality Model

mechanism may be needed. In Fig. 2, the execution of transition T_2 from cluster C_2 requires to place a token on the output place P_3 which belongs to cluster C_3. If C_2 and C_3 are assigned to different analysis hosts the token has to be forwarded via a communication channel.

An analysis host can be overloaded due to high CPU load and network traffic. This can hinder the effectiveness of the distributed analysis tasks. To avoid the problem, we allow the high-loaded host to move one of its atomic clusters to another host which has much lower load. The relocation takes into account the types of events analyzed by the moved cluster such that the benefits of grouping clusters are kept.

Figure 3 depicts the principal architecture of a distributed signature-based IDS applying EDL signatures for attack detection. A *sensor* logs relevant audit events from the observed traffic of a network component. The *atomic clusters* (*acs*) of an attack signature are assigned to *analysis units* (*AUs*) that are executed on various hosts. Since atomic clusters only require certain types of events, a *filter* forwards events to analysis units as needed. As described above, the assignment of atomic parts to two different analysis units (e.g., *AU1* and *AU2* in Fig. 3) which are located on different hosts demands for token forwarding. Thus, the analysis units exchange necessary state information (tokens) among each other using a local router (*rtr*) component. If required, e.g. for load balancing, an atomic cluster can be moved among hosts. The results of the distributed analysis are aggregated, e.g., in a database, labeled with a priority value, and evaluated by the security administrator.

3 Modeling IDS Core Functionality

To facilitate the engineering of the complex distributed IDS described in the previous section, we first only model the IDS core functionality. Here, we assume that all IDS analysis units are executed in a single host such that we do not need to cope with distribution aspects in the current design level.

The model-driven engineering method SPACE [Kra08] and its tool suite ARCTIS [KBH09] proved to be suited to handle coordination and communication between several components. SPACE uses hierarchies of UML activities that enable a complete and rather compact description of system behavior by graphical models. The models are collaborative which allows encapsulation of patterns facilitating a high degree of reuse. The major specification units of SPACE are *reactive building blocks*. Besides the UML activities, SPACE employs UML collaborations modeling the structure of systems and system parts, and special UML state machines called External State Machines (ESMs) to express the interface behavior of the blocks. The collaborative models can be model checked for design errors and automatically transformed to a component-oriented model in the form of UML state machines [KH07]. By diverse code generators, the component models can further be transformed to Java code running on various platforms.

The UML activity that models the IDS core functionality is depicted in Fig. 4. It is composed of three building blocks: *spa: Signature Parts Assigner* specifying a component that identifies signature parts and assigns each part to an atomic cluster instance, *ac: Atomic Cluster* modeling a generic atomic cluster (i.e., *ac* from Fig. 3) and *r: Router* describing the routing of EDL tokens among clusters (i.e., *rtr*). For each cluster, there is a separate instance of the block *Atomic Cluster* such that many instances are executed concurrently. This is signified as a shadow around *ac: Atomic Cluster*.

The semantics of UML activities is also based on token flows. On activity edges, activity tokens[1] passing from a node to the next can be used to model both control and data flows. The interaction between a block and its environment is described by pins. Arrows on a white background are starting and terminating pins which model the activation respective termination of the blocks. Arrows on a black background are streaming pins which describe flows of activity tokens while a block is active. At the start of the system execution, activity tokens are placed in the initial nodes (•). In Fig. 4, one activity token flows through the call operation action *getInitCnf* which retrieves information about the initial configuration of clusters. The configuration is forwarded within an activity token to the starting pin of the block *r: Router*. Simultaneously, another activity token flows to pin *start* of the block *spa: Signature Parts Assigner* which further iteratively emits an activity token via streaming pin *out*. This token contains a signature part in the form of the Java

[1] To distinguish these tokens from those in the EDL description of Sect. 2, we use the term *activity tokens*.

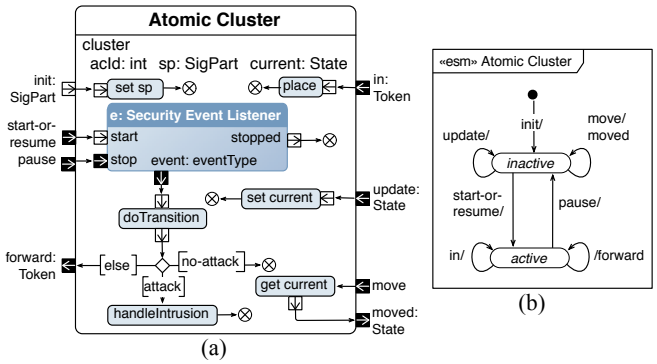

Figure 5: Atomic Cluster Model

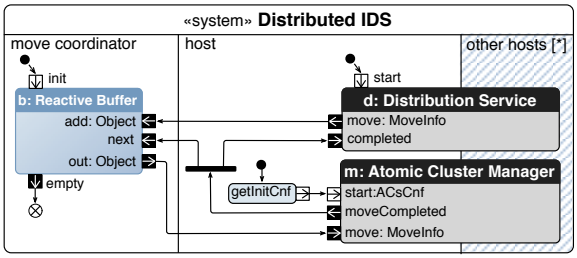

Figure 6: Distributed IDS Model

class *SigPart* and is sent to an instance of *Atomic Cluster* block identified by *acId*, as denoted in the select statement **select** acId. The block *spa: Signature Parts Assigner* later terminates and emits an activity token via pin *done*.

The externally observed behaviour of the *Atomic Cluster* block is described by its ESM depicted in Fig. 5(b). After being started by receiving an activity token via pin *init*, the block enters the state *inactive*. An activity token entered via pin *start-or-resume* triggers the transition to state *active*. As shown in Fig. 5(a), this makes the block ready to receive events from the *e: Security Event Listener* block which models the sensor component of an IDS. Following an activity token emitted via pin *event*, transition conditions are evaluated in the call operation action *doTransition*. There are three possible results expressed by a decision node: (1) *no attack*, since the transition in the EDL model leads to an escape place excluding an attack; (2) *attack*, when an attack is detected due to reaching an exit place which causes the execution of the call operation action *handleIntrusion*; (3) *else* that corresponds to an EDL token reaching a new interior place such that further analysis is needed and an activity token containing an EDL token is sent to another cluster via pin *forward*. Correspondingly, a cluster may also receive activity tokens containing EDL tokens via pin *in* from another cluster. The rest of the behavior will be described in Sect. 4 as it is only related to moving the responsibility of executing a cluster to another host.

In order to simplify the communication between clusters, all activity tokens are forwarded to the router (see Fig. 4) which further uses the configuration information to route them to the destination. Note that several pins of the blocks for the router and atomic cluster are not linked, since they will be used only in the distributed model described in the next section.

4 Modeling Distribution Functionality

Now, we extend the model of the IDS core functionality to the distributed IDS model by rearranging its building blocks and adding other blocks providing communication and handover of analysis functions between analysis hosts. To keep the synchronization efforts manageable, when transferring clusters, we first pause the analysis functions in all hosts, thereafter carry out a handover, and finally resume the analysis in the hosts again. Here, we assume that moves of clusters are relatively rare such that the temporary blocking of the analysis will not strain the overall performance too much.

The model of the distributed IDS is depicted in Fig. 6. It consists of many *collaborative* components. In particular, we have an arbitrary number of analysis hosts executing the various atomic clusters and a move coordinator harmonizing the transfer of analysis functions between the hosts. The functionality of the move coordinator is described in the activity partition of the same name on the left side of the activity in Fig. 6. The two other partitions describe the behavior of the hosts. This modeling feature is used to represent an arbitrary number of collaborating system components which all conduct the same behavior. The functionality of a representative host is described within the partition *host*, while the hatched partition *other hosts* is a placeholder for all the other hosts collaborating with this representative.

As shown in Fig. 6, each *host* takes part in *d: Distribution Service* which therefore spans both the partitions *host* and *other hosts*. This block specifies the distribution logic of the IDS system. It may send out an activity token via pin *move* to move an atomic cluster from one host to another whenever the host is overloaded. This command is then sent to a *move coordinator* that ensures that only one move is in progress at a time, queueing any other move command until the current move is completed. The queuing is realized by *b: Reactive Buffer*, a building block from the ARCTIS library. As we assume these moves occur rarely, it is unlikely that too many move commands will be queued up at the move coordinator.

The *Atomic Cluster Manager* manages local clusters in a host. When started, it receives data describing the initial configuration that tells the host which clusters are local to it. This block also handles the moving of a cluster if commanded to do so by receiving a token at pin *move*. After a move is finished, a notification is sent via pin *moveCompleted* to both the distribution service and the move coordinator to allow new commands to be carried out.

We give only a short summary of block *Distribution Service* in this paper, since its inner structure is rather similar to

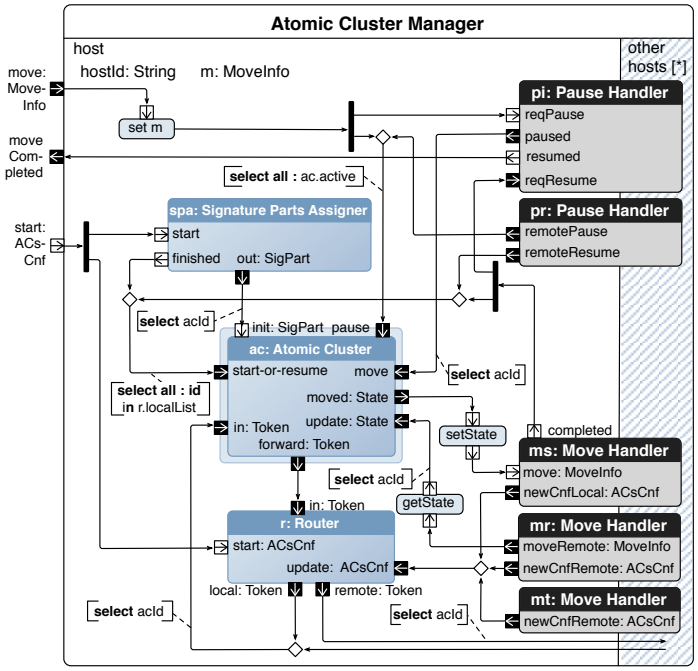

Figure 7: Atomic Cluster Manager Building Block

but also simpler than block *Atomic Cluster Manager* which will be later described in detail. The *Distribution Service* block contains a load monitoring component which periodically outputs a metric for the overall load of an IDS host. Whenever a host detects an overload situation, it will initiate a search for another host with a significantly lower load to handle the execution of an atomic cluster instance. If such a host is available, a move request along with the information necessary to carry out the transfer of the cluster instance is emitted from the block via pin *move* (see Fig. 6). The *completed* pin is used to signal the block that the previous move request is finished and thus a further search for another host may be executed. This is necessary, since the transfer process takes time to complete and only one search request in transit is allowed in order to minimize the communication cost.

The details of the *Atomic Cluster Manager* are shown in Fig. 7. This block applies blocks already used in the IDS model in Sect. 3. In addition, we create collaborations *Pause Handler* and *Move Handler* to deal with the synchronization and handover of analysis functions between analysis hosts. Atomic cluster manager is symmetric, since a host may initiate a pause request and at the same time it must be ready to receive a request from others. Therefore, the host in the *Atomic Cluster Manager* block needs to play both roles of *Pause Handler*. In instance *pi*, the host plays the role as the initiator, while in *pr* as the receiver. Likewise, we use three instances of *Move Handler* describing the roles an analysis host may play when a cluster is moved, i.e., a sender of a cluster (in instance *ms*), a recipient (instance *mr*) and other or in this case neither a sender or a recipient (instance *mt*). We discuss the block *Move Handler* in greater detail below.

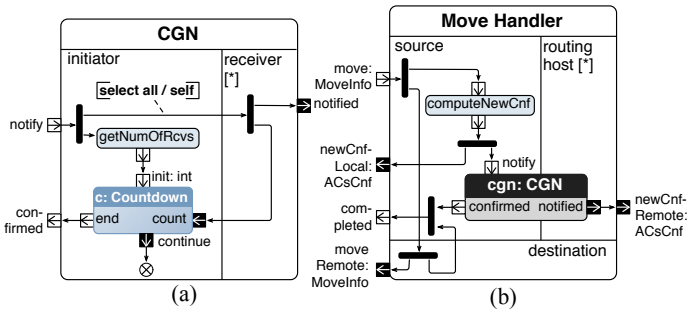

Figure 8: CGN and Move Handler Blocks

The atomic cluster manager maintains a set of clusters which are described by the block *ac: Atomic Cluster* and a shadow around it. Each analysis host has the same set of atomic clusters, but only those instances which are assigned to a particular host are active. Like in the model in Sect. 3, the block *spa: Signature Part Assigner* first assigns signature parts to a set of cluster instances via pin *out*. In parallel, the router is initialized with the initial atomic cluster's configuration. Since atomic clusters have to be assigned to a specific analysis host, the configuration also contains the mapping between clusters and analysis hosts. After initialization, the clusters that are assigned to the host are activated via pin *start-or-resume*, while the remaining stay inactive (see also Fig. 5(b)). The active clusters listen to security events as described in Sect. 3.

Whenever a move request is received, all active local atomic clusters are deactivated by sending a token to the pin *pause*. As depicted in Fig. 5(b), this brings all clusters to be in state *inactive*. Simultaneously, a pause request is sent to all other hosts via instance *p0* of *Pause Handler*. Since *Pause Handler* is basically a simpler version of the *Move Handler*, we omit here its detailed description but introduce block *CGN* (Confirmed Group Notification) depicted in Fig. 8(a), which realizes the coordination between the hosts. *CGN* which is included in the ARCTIS library for group communication sends a message to all receivers and then waits for all of them to acknowledge receipt of the message before terminating via pin *confirmed*. The other hosts receiving this pause request will pause their local active clusters. This is shown as an edge from pin *remotePause* of instance *pr: Pause Handler* in Fig. 7. After all clusters in all hosts are paused as indicated by a token emitted via pin *paused* in *pi: Pause Handler*, the current EDL state of the cluster to be moved is retrieved and sent within an activity token via pin *moved* to a destination cluster in another host. The receiving cluster will accept the EDL state via pin *update*.

The move is managed by block *Move Handler* whose behavior is modeled by the activity in Fig. 8(b). The source host computes a new configuration from the information about which atomic cluster to be transferred from this host to a given destination host. This configuration is further sent to all other routing hosts, while a copy is also sent back locally through the pin *newCnfLocal*. The activity token passing

from *source* to *destination* contains the EDL token setting of the moved cluster as data. When all routing hosts have acknowledged receiving the atomic cluster configuration and the destination host has acknowledged its message, the move handler collaboration terminates via pin *moveCompleted*.

As described in Fig. 7, once the move handler finishes its work, the host that initiated the move will request the others to resume their work via pin *reqResume* on *p0: Pause Handler*, which the other hosts receive via pin *remoteResume*. Thereafter, the IDS can continue detecting unwanted security attacks. When all hosts have acknowledged resume requests, a move completed notification is is forwarded via pin *moveCompleted* so that another move request may be carried out (see Fig. 6).

5 Validation and Evaluation

The formal semantics of SPACE enables direct checks of collaboration-oriented specifications for general design errors. So, it is possible to detect potential system flaws already at the modeling level. In particular, ARCTIS tool uses a model checker [KSH09] which can also simulate specifications to illustrate their behavior. Due to the compositional semantics of SPACE and the encapsulation via ESMs, it suffices to validate building blocks separately. This results in small state spaces which are analyzed for typical error situations, such as the non-conformance of an activity with its ESM and boundedness of communication. In addition, we check that an activity composed of several blocks obeys the external contracts of each block. For example, the activity of *Atomic Cluster* in Fig. 5(a) must conforms to its ESM depicted in Fig. 5(b). Furthermore, it must obey the ESM of *Security Event Listener* block.

In the case of the distributed IDS, the challenge with the collaborative building blocks lies in the fact that one IDS host communicates with potentially many others. In order to verify block *Atomic Cluster Manager* (see Fig. 7), instead of using the ESMs of blocks *Pause Handler* and *Move Handler*, we first create the ESMs of their instances, i.e., *pi*, *pr*, *ms*, *mr* and *mt*. This type of ESM does not describe the complete external behavior of a collaborative block, but only for a single role a symmetric block may play. Then, the model checker in ARCTIS uses these ESMs and the ESMs of *Signature Parts Assigner*, *Atomic Cluster* and *Router* to analyze the *Atomic Cluster Manager* block.

Another advantage of the SPACE method is the effective reuse of building blocks. As pointed out in [KH09], we reuse more than 70% of the system specification by reusing existing building blocks in the ARCTIS library. Due to encapsulation of the building block by the ESMs, we can thereby completely ignore the details of an activity. This is also referred to as compositional *black-box* reuse [FT96]. Furthermore, the reusable blocks may form distributed system patterns. With respect to our IDS example, the distribution functionality model is a pattern that can be used to develop any distributed systems which need to dynamically relocate partial tasks among components. In addition, the black-box type of reuse is especially helpful, when a system is created by different experts, since it confines different working tasks in the form of building blocks.

Table 1: Metrics for the IDS Core Model

Building Block	Complexity n	Reused	Experts
IDS Core [Fig. 4]	34		I + C
Distribute Signature Parts	23		I + C
Iterator	22	✓	
Atomic Cluster [Fig. 5(a)]	52		I + C
Security Event Listener	14		I + C
Router	29		I + C
Total complexity	174		

To give an estimation for the gains through the kind of reuse enabled by our method, we assign a complexity number n to each activity. This number is simply the sum of nodes n_{nodes} and edges n_{edges} within an activity and can be compared to measuring the lines of code in programming languages. For the *Atomic Cluster* block, for instance, we calculate a complexity of $n = 30\,nodes + 17\,edges = 47$. Table 1 and 2 list all complexity numbers for the building blocks involved in the IDS core functionality and distributed IDS models. To estimate the effort E in time and costs needed for building a system we multiply the complexity n with the number of expert groups involved. The unit of the result is not important, since we will only use it for comparison. The multiplication emphasizes that the main effort in creating a specification is not the actual writing, but the reading and understanding of it. Intuitively, this means that a specification leads to more effort (i.e., higher costs) the more complex it is and the more expert groups (implying personnel costs) have to handle it.

In the following we estimate the reduction of development effort enabled by the collaborative building blocks used by our method. For that, we compare two cases: one without and the other with building blocks. In our calculation we incorporate the different expert groups involved in the development, namely those for IDS systems (I) and those for general communication systems (C).

The overall complexity of the distributed system is $n = 599$. In a case without building blocks, *both* expert groups are involved in the entire specification. This means that both expert teams have to fully understand the specifications. For a distributed IDS that is constructed without the use of building blocks, we calculate therefore an overall effort of $E_{d1} = 599 \times 2 = 1198$.

In our actual development (*with* building blocks), we first developed the IDS core model which has an overall complexity of $n = 174$, as the sum of all building blocks in Tab. 1 implies. Since the *Iterator* with $n = 22$ can be reused, we only count $n = 152$ for the IDS core model. During its development both expert groups were involved, since the IDS experts needed guidance with the design method initially only known by the communication experts. This means an overall effort for the IDS core model of $E_c = 152 \times 2 = 304$. In order to add the distribution functionality to the IDS core

Table 2: Metrics for the Distributed IDS Model

Building Block	Complexity n	Reused	Experts
Distributed IDS [Fig. 6]	39		C
Reactive Buffer	39	✓	
Distribution Service	50		C
One	4	✓	
Load Monitoring	37		C
Component Monitor	10		C
Find Host With Minimal Load	32		C
Collected Group Response	26	✓	
Countdown	18	✓	
Atomic Cluster Manager [Fig. 7]	86		C
Router	29		I + C
Distribute Signature Parts	23		I + C
Iterator	22	✓	
Atomic Cluster	52		I + C
Security Event Listener	14		I + C
Pause Handler	13		C
Conf. Group Notif. [Fig. 8(a)]	21	✓	
Countdown	18	✓	
Move Handler [Fig. 8(b)]	27		C
Conf. Group Notif. [Fig. 8(a)]	21	✓	
Countdown	18	✓	
Total complexity	599		

model, only the involvement of the communication experts was needed. Furthermore, these experts did not have to build the entire system with a total complexity of $n = 599$ from scratch, but they could reuse building blocks from the existing libraries. The reused blocks from libraries added up to a complexity of $n = 187$, which means about 31%. Since a number of blocks could be taken from the IDS core model, the extension only involved blocks with a total complexity of $n = 294$.[2] The complete effort for creating the distributed system *with building blocks* can therefore be calculated as $E_{d2} = E_c + 1 \times 294 = 598$.

This means that the abstraction and reuse enabled by the reactive and collaborative building blocks can reduce the overall development effort by roughly 50%. Of course, these numbers are an estimation, based on a simple complexity metric introduced above. We argue, however, that the real gains are even more substantial than the 50%. In the calculation we have assigned both teams to the development of the IDS core model. In reality, this was rather the team of IDS experts together with one person familiar with *SPACE*, which would further reduce the effort E_c and E_{d2}. One could also argue that the effort from handling a building block with complexity n is not proportional to n but rather polynomial, which further increases the gains of reuse. In addition, our calculation did not take into account the incremental verification process. Since building blocks can be analyzed incrementally, the verification is virtually starting with the first blocks produced, and a high consistency throughout the entire development process is achieved.

[2]These are all blocks in Tab. 2 that are created by the C-experts only.

6 Related Work

The component-based engineering approach [BW96, RM01] has been proposed to lower the effort in developing complex systems. Such systems can be built from off-the-shelf components and thus reducing the amount of time needed to create new systems. Our method also utilizes reusable models from the ARCTIS library that helps with the development of complex, distributed systems. However in contrast to the component-based approach, we use collaboration-oriented models that explicitly specify the communication between several components. This models encapsulate the collaborative behavior patterns, such that they can handle the complexity of coordinating distributed functions comfortably.

The aspect-oriented approach [KLM+97, CT04] is a relatively new method to modularize the development of systems that consist of intertwined aspects. The crosscutting concerns of coordinating and management of distributing analysis tasks can be modeled as aspects which are woven into the specification of basic IDS functionality resulting in the distributed IDS design. However, the modifications caused by the aspect weaving may introduce new unintended behaviors. In our approach such behaviors can be avoided due to the complete, formal systems specification.

7 Final Remarks

In this paper we showed how to model a distributed and load-balanced IDS based on EDL signatures by using the collaboration-oriented SPACE method. This enabled us to handle the complexity of coordinating the handover of analysis tasks by utilizing the hierarchical structure of SPACE which specifies separate functions in different blocks. Furthermore, reusable building blocks from our library helped to cope with a fair number of communication and synchronization problems without the need to create new models. As shown in Sect. 5, this reduced the effort spent to design a complex distributed system. In addition, only engineers familiar with communication systems are needed during the development of the distributed model, since all functions related to IDS can be reused from the model describing the IDS core.

Besides the more general design properties described in Sect. 5, other distributed IDS specific properties also need to be checked. They can be stated as follows:

- All atomic cluster instances in all hosts are in state *inactive* (see Fig. 5(b)), when a handover of analysis functions is in progress.
- An EDL token is never routed to an inactive atomic cluster instance.
- Only one move is in progress at a time.
- Every atomic cluster is assigned to at most one host at all times.
- An atomic cluster is always eventually assigned to a host.

To verify also those properties, we currently extend our tool-support enabling the developers to specify application specific properties. Since the semantics of SPACE is based on the compositional Temporal Logic of Actions (cTLA) [HK00], we can further formulate and verify refinement relations, as the one that the distributed IDS model implements the model of the IDS core.

The distributed IDS model specified in Sect. 4 is created under the ideal assumptions that the IDS hosts never crash and messages are never lost. To ensure the dependability of such a system we must take into account the properties of the real execution environment. We plan to augment our system design with fault-tolerance mechanisms, e.g. to avoid system failures, even in the presence of errors like a crashed host. However, such mechanisms may greatly increase the complexity of the system, so that integrating them with the existing design can introduce new design faults. Hence, it is important to verify that the augmented system's functional properties are still fulfilled, both under ideal conditions and in the presence of errors that we intend to tolerate. We plan to achieve this by extending our verification approach (model checking) to handle different types of more realistic execution environment semantics.

References

[BW96] A.W. Brown and K.C. Wallnan. Engineering of component-based systems. *IEEE Int. Conf. on Engineering of Complex Computer Systems*, 0:414, 1996.

[CDK+09] Georg Carle, Falko Dressler, Richard A. Kemmerer, Hartmut König, Christopher Kruegel, and Pavel Laskov. Network attack detection and defense – Manifesto of the Dagstuhl Perspective Workshop. *Computer Science - R&D*, 23(1):15–25, 2009.

[CPV97] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with mobile code paradigms. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, 1997.

[CT04] Mariano Ceccato and Paolo Tonella. Adding distribution to existing applications by means of aspect oriented programming. 2004.

[FT96] William Frakes and Carol Terry. Software Reuse: Metrics and Models. *ACM Computing Surveys*, 28(2):415–435, 1996.

[HK00] Peter Herrmann and Heiko Krumm. A framework for modeling transfer protocols. *Computer Networks*, 34(2):317–337, 2000.

[Jen01] Nicholas R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.

[KBH09] Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Compositional service engineering with arctis. *Telektronikk*, 105(2009.1), 2009.

[KH07] Frank Alexander Kraemer and Peter Herrmann. Transforming collaborative service specifications into efficiently executable state machines. In *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 7 of *Electronic Communications of the EASST*. EASST, 2007.

[KH09] Frank Alexander Kraemer and Peter Herrmann. Automated encapsulation of uml activities for incremental development and verification. In *Proceedings of the 12th Int. Conference on Model Driven Engineering, Languages and Systems (Models)*, volume 5795 of *Lecture Notes in Computer Science*. Springer, 2009.

[KH10] Frank Alexander Kraemer and Peter Herrmann. Reactive semantics for distributed uml activities. In John Hatcliff and Elena Zucca, editors, *Formal Techniques for Distributed Systems*, volume 6117 of *Lecture Notes in Computer Science*. Springer, 2010.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. Springer, 1997.

[Kra08] Frank Alexander Kraemer. *Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks*. PhD thesis, Norwegian University of Science and Technology, August 2008.

[KSH09] Frank Alexander Kraemer, Vidar Slåtten, and Peter Herrmann. Tool support for the rapid composition, analysis and implementation of reactive services. *Journal of Systems and Software*, 82(12):2068–2080, December 2009.

[MSK05] Michael Meier, Sebastian Schmerl, and Hartmut König. Improving the Efficiency of Misuse Detection. In *DIMVA*, volume 3548 of *LNCS*. Springer, 2005.

[RM01] Marija Rakic and Nenad Medvidovic. Increasing the confidence in off-the-shelf components: a software connector-based approach. *SIGSOFT Softw. Eng. Notes*, 26(3):11–18, 2001.

[RWMB98] Diane T. Rover, Abdul Waheed, Matt W. Mutka, and Aleksandar M. Baqić. Software tools for complex distributed systems: Toward integrated tool environments. *IEEE Concurrency*, 6(2):40–54, 1998.