The Guided System Development Framework: Modeling and Verifying Communication Systems

Carvalho Quaresma, Jose Nuno; Probst, Christian W.; Nielson, Flemming

Published in:
Leveraging Applications of Formal Methods, Verification and Validation - Specialized Techniques and Applications

Link to article, DOI:
10.1007/978-3-662-45231-8_42

Publication date:
2014

Document Version
Peer reviewed version

The Guided System Development Framework: Modeling and Verifying Communication Systems

Jose Quaresma, Christian W. Probst, Flemming Nielson

Technical University of Denmark
{jncq,cwpr,fnie}@dtu.dk

Abstract. In a world that increasingly relies on the Internet to function, application developers rely on the implementations of protocols to guarantee the security of data transferred. Whether a chosen protocol gives the required guarantees, and whether the implementation does the same, is usually unclear. The Guided System Development framework contributes to more secure communication systems by aiding the development of such systems. The framework features a simple modelling language, step-wise refinement from models to implementation, interfaces to security verification tools, and code generation from the verified specification. The refinement process carries thus security properties from the model to the implementation. Our approach also supports verification of systems previously developed and deployed. Internally, the reasoning in our framework is based on the Beliefs and Knowledge tool, a verification tool based on belief logics and explicit attacker knowledge.

1 Introduction

Developing secure communication systems is difficult. Application developers rely on the implementations of protocols to guarantee the security of data transferred. Whether a chosen protocol gives the required guarantees, and whether the implementation does the same, is usually unclear.

Verifying secure communication systems is difficult, too, though for different reasons. While a plethora of formal approaches and tools for protocol verification exist, they are often not accessible to developers, and only connect the guarantees to the implementation of the protocol.

The Guided System Development (GSD) framework aims at making development and verification of secure communication systems easier by bridging the gap between system development and verification of communication protocols. The knowledge and skills required to successfully use a security verification tool are significant. With the GSD framework [1, 2] it is possible to use such tools and have access to their results *without* the need for that specific knowledge.

The main achievement of the GSD framework is to make building secure communication systems the *only* option. This is reached through a number of components: a simple and intuitive modelling language, step-wise refinement of guarantees from the model to its implementation, built-in interfaces to established security verification tools, and finally code generation.

The rest of the paper is structured as follows: after an introduction of a running example we use throughout the paper, Sec. 2 discusses related work. In Sec. 3 we present an overview of the framework. We then introduce the Abstract Global level in Sec. 4 and the Concrete level in Sec. 5. In Sec. 6, we present the interface to verification tools and code output. We evaluate the tool in Sec. 7 by presenting its usage on the modelling and verification of the Automatic Dependent Surveillance-Broadcast (ADS-B) system, a new air traffic management surveillance system that is replacing the radar as the main means for traffic management in the near future. Finally, Sec. 8 concludes the paper and discusses future work.

1.1 The Message Board

To illustrate the use of GSD, we use the communication behind a message board as an example. This message board allows a user to share a message either anonymously and/or confidentially. The combination of these properties gives rise to four different kinds of messages:

1. The message sent can be seen by everybody without any guarantees regarding the identity of the sender,
2. The message sent can be seen by everybody and it has guarantees regarding the identity of the sender,
3. The message sent can only be seen by a particular user without guarantees regarding the identity of the sender, or
4. The message sent can only be seen by a particular user and it has guarantees regarding the identity of the sender

For space reasons, we only model sending a message in an authenticated way, *i.e.*, that other users have guarantees regarding the message's author.

2 Related Work

CaPiTo [3] connects the abstract specification of Service-Oriented Systems with the standard protocol suites used in industry in an independent way, *i.e.*, the description of the system in terms of messages exchanged is separated from the description of which standard suites are going to be used. This separation of concerns when modelling a Service-Oriented System greatly inspired our framework. Furthermore, CaPiTo also allows the verification of the modeled protocols and the code generation of parts of the system. This is accomplished by providing a specification language, means to verify the security of the specified protocol, and the translation from the protocol specification language into an executable language. Compared with CaPiTo, our modelling language is simpler and more intuitive, it has views of different levels of the system, and connects to more verification tools, including a GSD-specific tool introduced in Sec. 6.1.

AVANTSSAR aims at validating trust and security of Service-Oriented Architectures. Compared with the GSD framework, the AVANTSSAR project[1] requires a higher level of expertise to start using it and does not provide automatic code generation, functionality that we believe is important when providing tools to help system designers and programmers.

A similar tool to the GSD framework is the Automatic Generation, Verification and Implementation of Security Protocols (AGVI) toolkit [4], that allows the system designer to describe the security requirements and the system specification. In AGVI a *protocol generator* creates candidate protocols that satisfy the given system requirements. After that, the protocols are analysed [5] and the ones that do not satisfy the desired security properties are discarded. Finally, a code generator translates the formal protocol specification into Java code. In comparison to AGVI, the GSD modelling language focusses on properties of communication. Furthermore, the GSD frameworks can by design be extended new formal verification tools, new output languages, new abstract ways of representing message security properties in the modelling language (extending the security modules presented in Sec. 4.2), and also new ways of implementing the different security modules.

There also exist more targeted work on modeling and implementation of secure communication systems. When compared to the GSD framework, these tools usually feature more concrete and complex modelling languages and target a specific verification tool. FS2PV [6] derives a formal model from a protocol code written in F (a first-order subset of F#) and symbolic libraries. The translation is made to π-calculus, which can then be verified by ProVerif [7]. Swamy *et al.* [8] developed a dependently typed language (F*) aimed at secure distributed programming. Programs written in F* are translated to .NET bytecode. Other more recent work [9] enables the verification of a protocol with CryptoVerif [10] and then translates that specification into OCaml.

3 Framework Overview

The overall structure of the GSD framework is shown in Fig. 1. The framework is composed by three levels that represent different levels of abstraction. System developers specify the desired system at the *Abstract Global* level, using security modules (Sec. 4.2), to specify the required security assurances for the data being exchanged in the modeled system.

By unfolding the security modules using plugins, the Abstract Global level is transformed into a model of the system in the *Concrete Global* level. Plugins connect the abstract modules of the Abstract Global level with their implementations. For instance, if there is a security module in the Abstract Global specification that requires some data to be sent in a confidential way, this translation would replace it with an implementation of, for example, the Transport Layer Security (TLS) cryptographic protocol. The use of plugins, which is similar to the work by Gao *et al.* [3], separates the desired security assurances of the

[1] `http://www.avantssar.eu`, last accessed May 2014.

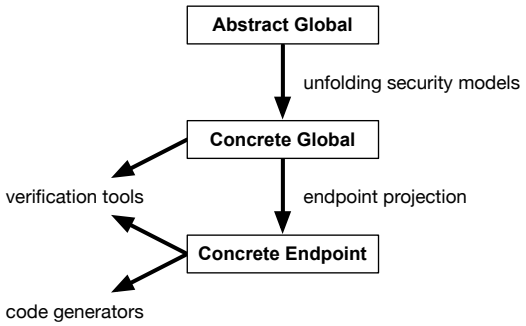

Fig. 1. Overview of the Guided System Development framework.

exchanged data and the way to provide those assurances. In Sec. 5.1, we discuss the use of plugins and the flexibility that they provide.

A specification at the Concrete Global level is made more concrete using endpoint projection, resulting in the *Concrete Endpoint level*. This step separates the specifications for each individual, and is closer to the final implementation and to some of the verification tools (Sec. 5.2).

Contracts, another component of GSD, represent the desired outcomes of the different security modules in the Abstract Global level; they describe the modules' semantics. For example, part of the contract for the confidentiality module expresses that only the intended recipient of a message sent confidentially is able to read the message. Contracts enable some preliminary reasoning in the Abstract Global level, and the verification of the implementations of each security module by comparing the desired outcomes with the outcomes achieved by the different implementations.

3.1 Framework Inputs and Outputs

The modeling language used in GSD is strongly influenced by Alice and Bob notation, a simple and intuitive way of modelling communication system. The example in Fig. 2 uses this notation to model a message `msg` being sent by a principal called `User` to another principal called `Board`. The complete language syntax for the input language (in the Abstract Global level) is presented in Fig. 3.

When building a system with secure communications using GSD, the input for the framework is the specification of the system in the Abstract Global level, which includes security assurances for the exchanged messages. The language at this level is simple and intuitive and when using it, the step-wise refinement will lead to an implementation that provides the specified security assurances.

```
User → Board : msg
```

Fig. 2. Example of sending a message from `User` to `Board` in Alice and Bob notation.

$$
\begin{aligned}
\text{system} &::= \text{stm;} \mid \text{system stm;} \\
\text{stm} &::= \text{principal} \rightarrow \text{principal} : \text{msgs} \\
\text{principal} &::= \textit{string} \\
\text{msgs} &::= \text{msg} \mid \text{msgs, msg} \\
\text{msg} &::= \text{el} \mid \text{secModule} \\
\text{el} &::= \textit{string} \\
\text{secModule} &::= \text{secAssurance(args)} \\
\text{secAssurance} &::= \mathit{Auth} \mid \mathit{freshAuth} \mid \mathit{Conf} \mid \mathit{Sec} \mid \mathit{freshSec} \\
\text{args} &::= \textit{string} \mid \text{args, } \textit{string}
\end{aligned}
$$

Fig. 3. Syntax of the language in the Abstract Global level.

It is, however, also possible to use the framework by writing the specification in one of the other two abstraction levels presented before. If so, one would not take full benefit of GSD, but would still benefit from some of the connections to the verification tools and code outputs. We believe that this option is useful for more experienced system developers or for already implemented systems, in which case a specification closer to the implementation might be easier to write.

The outputs of GSD can be divided into two categories: the information from the supported verification tools and the implementation of the modeled system. We present these in more detail in Sec. 6.

4 Abstract Global Level

The goal at this level is to provide the developer with a simple and intuitive language that has the necessary tools to model the communication system that is being developed. As shown in Fig. 3, and as mentioned in Sec. 3.1, this language is similar to the Alice and Bob notation but extends that notation with security modules, which are presented in Sec. 4.2.

The logic used to express (and reason about) security properties is based on BAN logic [11] and more generally on SVO logic [12], a logic that unifies several different belief logics, including BAN logic itself. We use BAN and SVO to reason about the beliefs of the principals involved in the different message exchanges. Based on the beliefs at the end of a series of message exchanges, we are able to argue about security properties of the exchanged messages.

4.1 The Logic

BAN and SVO logics focus on the beliefs that legitimate principals are able to infer from a message exchange, and target authentication. Such logics are not less suited to directly reason about confidentiality, since confidentiality concerns what some non-legitimate principal might, or might not, be able to see from a

P Received el - principal P received an Element el;
P Sees el - principal P sees a specific Element el;
P Believes t - principal P believes in a specific Term t;
P Said el - principal P said, at some point in time, a specific Element el;
P Says el - principal P recently said the Element el;
Conc(el1,el2) - concatenation of two Elements;

Fig. 4. Terms in our logic.

message exchange. There are several approaches to handle confidentiality in this case. We extend belief logics with explicit reasoning about the knowledge of non-legitimate principals: the beliefs they are able to infer from the message exchange and what they are able to see and (most importantly for the confidentiality reasoning) what they *not* are able to see about the exchanged messages. The chosen approach results in a simple model that is easy to reason about.

We consider non-legitimate principals to be Dolev-Yao attackers [13], *i.e.*, they are not only able to see all the exchanged messages but also capable of initiating protocol communications with legitimate principals and to forge new messages based on acquired knowledge.

Our logic is composed by principals and elements (all the artifacts that can be sent from one principal to another). The different terms in our logic are shown in Fig. 4. The rules used to reason about this logic are introduced in Sec. 6.1.

```
User → Board : Auth(User, message)
```

Fig. 5. Specification of `User` sending a message to the `Board` in an authenticated way.

4.2 The Security Modules

The most important elements at the Abstract Global level are the security modules, which model security assurances for data exchanges:

- **None** is not actually a security module, but sends data in plaintext.
- **Auth** sends data such that the receiver can identify the sender.
- **Strong Auth** adds freshness to the Auth module to prevent replay attacks.
- **Conf** sends data sent such that only the intended receiver can read it.
- **Sec** is the conjugation of the Auth and the Conf modules.
- **Strong Sec** is the conjugation of the Strong Auth and the Conf modules.

Fig. 5 shows how to model the authenticated message sent by `User` to `Board`.

4.3 Semantics of the Security Modules

In GSD contracts are attached to the security modules on the Abstract Global level, describing the results of using the different modules. This can be used to

define module semantics and to verify different implementations of a module by checking the specified security properties against the respective implementations.

Before defining the contracts, we briefly discuss the use of integrity in GSD. Integrity can have different meanings depending on the field it is being used in, and can even have slightly different definitions in the same field. Here, we consider integrity to mean that a message is not corrupted over time or in transit [14]. For message exchange, integrity is guaranteed when the contents of a message cannot be changed in transit without changes being instantly observable by the recipient. In the GSD framework we assume integrity in all exchanged messages, *e.g.*, by sending a signed digest of the message together with the full message. With integrity, we have the following rule (where P *sees* m means that any principal is able to see m):[2]

$$\frac{X \rightarrow Y : m}{P \ sees \ m}$$

We can now present the rules for the different security modules. Please note that for the sake of space, we only present the more simple Auth and Sec modules.

Authentication. When a principal sees $Auth(X, w)$, he knows that the message was sent by X, but knows nothing about the freshness of the message:

$$\frac{Z \ sees \ Auth(X, w)}{Z \ sees \ w, Z \ believes \ X \ said \ w}$$

Confidentiality. A message that is confidential to X can only be read by him:

$$\frac{Z \ sees \ Conf(X, w), \quad Z \ is \ X}{Z \ sees \ w}$$

Security. The security module combines authentication and confidentiality:

$$\frac{Z \ sees \ Sec(V, X, w), \quad Z \ is \ X}{Z \ sees \ w, Z \ believes \ V \ said \ w}$$

Applying the rules presented above to the conjugation of the authentication and the confidentiality modules results in the same beliefs that were presented above for the security module:

$$\cfrac{\cfrac{\cfrac{\cfrac{X \rightarrow Y : Conf(Y, Auth(m))}{P \ sees \ Conf(Y, Auth(X, m))}}{Y \ sees \ Conf(Y, Auth(X, m)), \ Y \ is \ Y}}{Y \ sees \ Auth(X, m)}}{Y \ sees \ m, \ Y \ believes \ X \ said \ m}$$

[2] We extend BAN and SVO logics with principal variables P that represent all the principals that see the messages being exchanged.

5 The Concrete Levels

There are two concrete levels in GSD: the Concrete Global level (Sec. 5.1) and the Concrete Endpoint level (Sec. 5.2). The model of the system on these levels is closer to the languages used by the verification tools and the implementation, but not as simple and intuitive as the one in the Abstract Global level.

5.1 Concrete Global Level

We obtain the Concrete Global level by unfolding the different modules at the Abstract Global level. The different plugins for each of the different security modules and represent implementations of the corresponding security module, for example, implementations using TLS, WS-Security, or a Public Key Infrastructure (PKI). As previously mentioned, it is possible to verify the chosen implementation for a security module by checking that it satisfies the respective contract.

From this level the GSD framework interfaces with the Beliefs and Knowledge tool (Sec. 6.1) and the Open-Source Fixed-Point Model-Checker (Sec. 6.3). s.

For our example system we choose to implement the Auth module using a PKI infrastructure. The result of applying that plugin to the Auth module is shown in Fig. 6.

5.2 Concrete Endpoint Level

In order to translate from the Concrete Global level to the Concrete Endpoint level, we apply an endpoint projection [15]. This technique extracts the views of the different principals present in a specification of the global view of the system. The resulting model at this level has the views of the different principals that participate in the communication system.

This translation is performed by going through the model with a global view of the system and, for each of the actions in the model, generating the correspondent actions that are performed by the different principals. For example,a message being sent from `A` to `B` in the global view, is projected to the independent specification of the correspondent actions of `A` and `B`, *i.e.*, `A` would send the message and `B` would receive it.

The model of our message board example at this level is shown in Fig. 7.

6 Verification Tools and Code

In this section we present the formal methods tools that GSD currently interfaces with. The Beliefs and Knowledge tool (Sec. 6.1) was developed as part of the

```
User → Board: User, Encryption(message, PrivKey(User));
```

Fig. 6. An authenticated message of the example system in the Concrete Global level.

```
1    User:
2      send(Board, (User,Encryption(message, PrivKey(User))))
3    Board:
4      receive(User,(User, Encryption(message, PrivKey(User))))
```

Fig. 7. An authenticated message of the example system in the Concrete Endpoint level.

- **A $\longrightarrow$ B : m $\Longrightarrow$ P Received m** - When a message is sent between two principals, every principal with access to the Ether will receive that message.
- **A Received el $\Longrightarrow$ A Sees el** - If a principal receives an Element, he is able to see it (note that the principal might be able to see the Element, but not what is inside it).
- **A Sees Enc(el,privKey(P)) $\wedge$ A Sees pubKey(P) $\Longrightarrow$ A Sees el $\wedge$ A Believes P Said el** - If principal A sees a message encrypted with another principal's private key and if A has access to the correspondent public key then A can see the encrypted element and also knows who sent it.

Fig. 8. Examples of the systems predefined rules.

GSD framework. The GSD framework outputs the system model to code by replacing the different elements of the Concrete Endpoint specification with predetermined Java blocks implementing those elements.

6.1 The Beliefs and Knowledge Tool

The Beliefs and Knowledge tool (BAK) verifies the security of communication protocols by reasoning about the beliefs and the knowledge that the different principals involved in a communication system acquire throughout message exchange. The tool uses the Z3 SMT Solver [16] and adds an extra layer that facilitates the modeling of message exchanges and the reasoning about those messages.

Predefined system rules. The extra layer defined in the BAK tool is composed of predefined system and inference rules specifying how principals construct and read the exchanged messages, how they acquire the different beliefs, etc. Some examples of these rules are shown in Fig. 8. A and B represent specific principals, P represents any principal, m represents a message in plaintext, $Enc(el, k)$ represents the encryption of the element el with the key k, and $pubKey(x)$ and $privKey(x)$ represent the public and private keys of principal x.

The third rule of Fig. 8 specifies which beliefs a principal can infer from a message encrypted with a private key: if the principal knows the correspondent public key, then he is able to decrypt it, see the element that had been encrypted, and have assurances on which principal encrypted the element. In Fig. 9, that rule is presented in SMT-LIB2.

There are two inputs for the BAK tool: a model (M) of the system we want to analyse and the goals we want to verify. Given these, the tool tests each goal

```
(assert (! (forall ((x Principal) (w Principal) (el Element))
      (! (=> (and (Sees x (EncModule el (PrivKey w)))
                  (Sees x (PubKey w)))
             (and (Sees x el)
                  (Believes x (Said w el))))
       )
       :pattern ((Sees x (EncModule el (PrivKey w))))
      )
     )
     :named privKeyDecryption)
)
```

Fig. 9. One of the system rules regarding decryption.

```
(MsgSent User Board (Conc User (EncModule message (PrivKey User))))
```

Fig. 10. An authenticated message of the example system modeled in SMT-LIB2.

(*goal*) against the modeled system. One implication of the way SMT works and the way we are modelling the system rules and the system itself is that we need to test the goals in the negated form. Both in the set of predefined system and inference rules (R) and in the system model (M) we only assert positive facts. When testing a goal in the positive form, the SMT solver will always find a model where the goal would be satisfiable since there will never be a negative rule to contradict the goal in the positive form. On the other hand, when testing a goal in the negative form it might contradict one of the assertions that are derived from $R \wedge M$, which can be interpreted as all the knowledge, *i.e.*, assertions, that can be derived from the system model. In that case the result will be unsatisfiable. If the negated goal does not contradict any of the assertions that are derived from $R \wedge M$, then the result will be satisfiable.

Therefore, we use $R \wedge M \wedge (\neg goal)$ to verify the system. If the system is satisfiable, then there is a representation of $R \wedge M$ where $\neg goal$ holds, which tells us that *goal* is not an assertion derived by $R \wedge M$. Due to the way we model the system and the system rules, this means that *goal* does not hold in the system. On the other hand, if the system is unsatisfiable, then there is no interpretation of $R \wedge M$ where $(\neg goal)$ holds. That can only happen if *goal* is derived from $R \wedge M$, which means that *goal* holds in the system. So, if any of the original goals we want to test is in the positive form we negate it before performing the test and interpret the result given by the tool according to that.

Fig. 10 shows the model of the authenticated message in SMT-LIB2, a standard format accepted as input by several SMT solvers, including Z3.

Tool Outputs. When analyzing $R \wedge M \wedge (\neg goal)$, the BAK tool does not only return satisfiability but also provides extra information that helps understanding and analysing the obtained results. If the system and the goal being analysed are satisfiable, then the tool also returns the representation that satisfies the assertions. On the other hand, if system and goal are unsatisfiable, the tool returns the unsatisfiability core, *i.e.*, a small set of assertions that make the system unsatisfiable. This set is not guaranteed to be minimal, but it provides useful

```
unsat (privKeyDecryption)
```

Fig. 11. Output of the BAK tool for the example system.

```
1 <User, Board, User, {| message |}:K_User->. 0
2 |
3 (Board, User; board1, board2).decrypt board2 as {|; message |}:K_User+ in 0
```

Fig. 12. The automatically generated LySa code of the authenticated message in the example system.

information regarding the system and the analysis result. Extracting information from a satisfiable model is not as simple as extracting information from the unsatisfiability core since the model tends to be complex and not easily readable.

For the example system the implementation of our authentication message should give guarantees regarding the authenticity of the message. It is possible to test this by verifying that `Board` knows that the `message` was sent by `User`, which is specified as `Board believes (User said message)` in belief logic. The result of applying the BAK tool to this goal in the negative form is shown in Fig. 11.

The first line tells us that the system model together with that negated goal is unsatisfiable, which mean that the goal we wanted to verify holds. The second line of the output is the unsatisfiability core returned by Z3. It is the name of the rule shown in Fig. 9 and it tells us that the `Board` obtained the belief specified in the goal by decrypting the `message`.

6.2 LySatool

The LySatool [17] performs security analyses of protocols described in LySa [18]. The tool performs a static analysis of the LySa specification of the protocol in the presence of a Dolev-Yao attacker [13]. The LySatool is implemented in the Standard ML (SML) functional programming language and it starts by encoding the analysis into a proper constraint language and then uses Succinct Solver [19] to compute the least solution to those constraints.

The LySa code that is generated by our framework is shown in Fig. 12.

6.3 The Open-Source Fixed-Point Model-Checker

The Open-Source Fixed-Point Model-Checker (OFMC) [20] is a symbolic security protocol analyser that detects attacks on the protocol and performs a bounded session verification by exploring the transition system of the protocol representation. Its primary input language is the Intermediate Format (IF) [21] specification, which describes a security protocol as an infinite-state transition system using set rewriting. The tool also accepts AnB [22] as input, a language similar to Alice and Bob notation, which is then automatically translated to IF, defining a formal semantics for AnB in terms of IF. OFMC uses several techniques that significantly reduce the search space of a protocol without introducing, or excluding, any attacks. Two of the major used techniques are *lazy*

```
User → Board : User,{message}inv(pk(User));
```

Fig. 13. The automatically generated AnB code of the authenticated message in the example system.

intruder and *constraint differentiation*. The first is a symbolic representation of the intruder while the latter is a general search-reduction technique. In Fig. 13, one can see the part of AnB code that corresponds to our authenticated message.

7 Evaluating GSD on ADS-B

In a recent evaluation, GSD was used to model and verify the Automatic Dependent Surveillance-Broadcast (ADS-B) [3] system [23], an air traffic management surveillance system that is being deployed with the intent of replacing the radar as the main system for airspace traffic management. This section first introduces the current implementation of ADS-B and then presents the way GSD was used to verify ADS-B.

Fig. 14. Overview of the ADS-B system.

7.1 The ADS-B System

ADS-B is a large wireless network, composed by ground stations and aircrafts that communicate with each other: the aircrafts report flight information (such as their position, velocity, and intent) and receive traffic and other information from the ground stations, as shown in Fig. 14. The main benefit of ADS-B is the provided higher accuracy regarding the aircraft position, which is crucial in an airspace where the aircraft density is increasingly higher.

The legitimate agents taking part in the ADS-B communication system are the aircraft and the ground-stations. ADS-B has two components that allow

[3] ADS-B General Information, `http://www.faa.gov/nextgen/implementation/portfolio/trans\_support\_progs/adsb/general/`, last accessed May 2014.

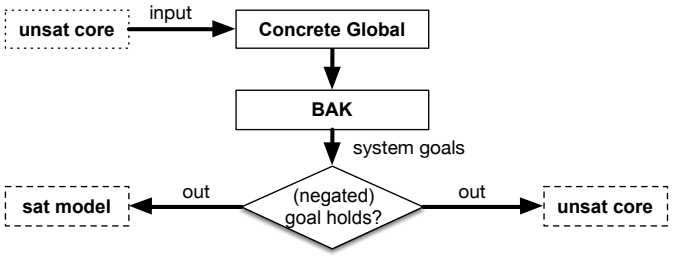

Fig. 15. The GSD framework applied to ADS-B.

these agents to communicate: ADS-B Out and ADS-B In. ADS-B Out consists of the messages that are broadcasted by the aircraft and ADS-B In concerns the capability of receiving the ADS-B Out messages. A message contains the aircraft's position and speed (both acquired through a positioning system, presently GPS) and potentially other information, such as intent. These broadcasted messages are received by the ground-stations and, in case ADS-B In is being used, the former will also be received by the aircraft that are within range of the broadcaster aircraft. As explained above, an aircraft is only capable of receiving air-to-air ADS-B messages broadcasted from other aircraft if it has equipment that provides ADS-B In capabilities. Another part of ADS-B In is the information broadcasted by the ground-stations. This will consist of traffic and weather information. During the transitional phase, the traffic information will have a mixture of ADS-B and Radar information, enabling the ADS-B In equipped aircraft to have a full view of the airspace surrounding it.

7.2 Applying GSD to ADS-B

The GSD framework was used to model and analyse the current implementation of ADS-B and its potential extensions as shown in Fig. 15. The most abstract level of the framework (Abstract Global) was not used, since that level is targeted for developing secure systems from scratch and not for modelling and analysing systems previously developed. Furthermore, the Concrete Endpoint level was not used either, since there was no interest in code, and the translation to the used tool to analyse our model is made from the Concrete Global level.

The ADS-B system model was used as input to the Concrete Global level, which was automatically translated into a language that can be used by the BAK tool (presented in Sec. 6.1) to verify the system and its properties (or goals), which are the other input to the framework.

GSD enabled the formal verification of the built-in security of the ADS-B communication system and reported that the system provides no authentication or confidentiality. GSD also enabled the security verification of the extensions suggested by Valovage *et al.* [24, 25] and, in this case, it reported that authentication and confidentiality were provided in their respective extensions.

8 Conclusion

The Guided System Development framework aims at helping developers building secure communication systems. It does so by enabling the modelling of systems in a simple and intuitive language, its verification by connecting that model to different formal verification tools, and translating it to code. In this paper we presented the capabilities of the GSD Framework by discussing the process of modelling, verifying, and implementing an authenticated broadcasted message. We also introduced the Beliefs and Knowledge tool, which extends belief logics with explicit attacker knowledge and uses the Z3 SMT Solver to enable the verification of security properties of communication systems. Furthermore, we presented an evaluation of a new airspace navigation system and its proposed extensions using GSD to model and verify the system's communications.

We believe that this framework represents a big step towards closing the gap between systems development and verification, but more work is necessary: We aim at extracting more information from the satisfiability model return by the BAK tool in order to provide more complete feedback to the developer, finalising the interfaces with LySatool and OFMC, and optimising the overall tool integration so that it is easier to use for the system developers. We also work on interfacing to and integrating the results from more analysis tools.

8.1 Acknowledgements

We would like to thank Roberto Vigo, Sebastian Mödersheim (both from the Technical University of Denmark), and Kristin Y. Rozier (from NASA Ames Research Center) for many fruitful discussions.

References

1. Quaresma, J., Probst, C.W., Nielson, F.: The Guided System Development Framework. In Pettersson, P., Seceleanu, C., eds.: Proceedings of the 23rd Nordic Workshop Programming Theory, Västerås, Sweden (October 2011) 69–72
2. Quaresma, J.: On Building Secure Communication Systems. PhD thesis, Technical University of Denmark (2013)
3. Gao, H., Nielson, F., Nielson, H.: Protocol Stacks for Services. In: Foundations of computer security. (2009)
4. Song, D., Perrig, A., Phan, D.: Agvi—automatic generation, verification, and implementation of security protocols. In: Computer Aided Verification, Springer (2001) 241–245
5. Song, D.X., Berezin, S., Perrig, A.: Athena: a novel approach to efficient automatic security protocol analysis. Journal of Computer Security **9**(1) (2001) 47–74
6. Bhargavan, K., Fournet, C., Gordon, A.D., Tse, S.: Verified interoperable implementations of security protocols. ACM Transactions on Programming Languages and Systems (TOPLAS) **31**(1) (2008) 5
7. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001. (2001) 82–96

8. Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: Proceeding of the 16th ACM SIGPLAN international conference on Functional programming. ICFP '11, New York, NY, USA, ACM (2011) 266–278
9. Cade, D., Blanchet, B.: From computationally-proved protocol specifications to implementations. In: International Conference on Availability, Reliability and Security (ARES), 2012, IEEE, 65–74
10. Blanchet, B.: A computationally sound mechanized prover for security protocols. Dependable and Secure Computing, IEEE Transactions on **5**(4) (2008) 193–207
11. Burrows, M., Abadi, M., Needham, R.: A logic of authentication. ACM Trans. Comput. Syst. **8** (February 1990) 18–36
12. Syverson, P.: A unified cryptographic protocol logic. Technical report, DTIC Document (1996)
13. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory **IT-29**(2) (1983) 198–208
14. Cullen, C.T., Hirtle, P.B., Levy, D., Lynch, C.A., Rothenberg, J., President, C.T.C.I.: Authenticity in a digital environment (2000)
15. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: Programming Languages and Systems. Springer (2007) 2–17
16. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In Ramakrishnan, C., Rehof, J., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 4963 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 337–340
17. Buchholtz, M.: User's Guide for the LySatool version 2.01. DTU. (April 2005)
18. Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Nielson, H.: Static validation of security protocols. Journal of Computer Security **13**(3) (2005) 347–390
19. Nielson, F., Nielson, H., Sun, H., Buchholtz, M., Hansen, R., Pilegaard, H., Seidl, H.: The succinct solver suite. Tools and Algorithms for the Construction and Analysis of Systems. 10th International Conference, TACAS 2004. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004. Proceedings (Lecture Notes in Computer Science Vol.2988) (2004) 251–265
20. Mödersheim, S., Viganò, L.: The open-source fixed-point model checker for symbolic analysis of security protocols. In: Foundations of Security Analysis and Design V. Volume 5705 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2009) 166–194
21. AVISPA: Deliverable 2.3: The intermediate format. `http://www.avispa-project.org` (2003)
22. Mödersheim, S.: Algebraic Properties in Alice and Bob Notation. In: International Conference on Availability, Reliability and Security (ARES), 2009, IEEE, 433–440
23. RTCA: DO-242A: Minimum Aviation System Performance Standards for Automatic Dependent Surveillance Broadcast (ADS-B). Technical report, RTCA (2002)
24. Valovage, E.: Enhanced ADS-B Research. In: 25th Digital Avionics Systems Conference, 2006 IEEE/AIAA. (oct. 2006) 1 –7
25. Viggiano, M., Valovage, E., *et al.*: Secure ADS-B Authentication System And Method (October 12 2007) WO Patent 2,007,115,246.