TUCS

Johan Lilius | Ricardo J. Machado
Dragos Truscan | João M. Fernandes (Editors)

Proceedings of MOMPES'05

2nd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS General Publication
No 39, May 2005

TUCS

Proceedings of

MOMPES'05

2nd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software

June 6, 2005, Rennes, France

Editors:

Johan Lilius
Ricardo J. Machado
Dragos Truscan
João M. Fernandes

Introduction

Welcome to the 2nd edition of the International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES 2005) held in Rennes (France), June 6th, 2005.

The Object Management Group's Model Driven Architecture (MDA) paradigm is an approach to the development of software, based on the separation between the specification of the systems and their implementation using specific platforms. This workshop focuses on the scientific and practical aspects related with the adoption of Model Driven development (MDD) methodologies (notation, process, methods, and tools) for supporting the construction of pervasive and embedded software. Suggested areas of interest in the workshop include, but are not restricted to:

- Specification of Platform Independent Models (PIMs) and Platform Specific Models (PSMs)
- PIM to PSM transformations
- MDD process for embedded and pervasive software
- Automatic code generation in MDD contexts
- Testing and validation in MDD contexts
- Tools for MDD of embedded and pervasive software
- Case studies on the application of MDD

We would like to thank the authors that submitted papers, the PC members for their excellent work in reviewing the papers, the ACSD'05 organizers, especially Laure Petrucci, for help in the local arrangements of this workshop, and TUCS and CREST for support in publishing the proceedings. Finally, we acknowledge the "Nordic Journal of Computing" for having accepted to publish a special edition with a selection of (extended and revised) versions of papers presented during the workshop.

MOMPES series of workshops proves once more to be a pertinent initiative to be able to get together people from the application domain of embedded and pervasive software and from the scientific community of model transformation.

We hope you all will enjoy the workshop!

The organizers:
Johan Lilius, TUCS (FI)
Ricardo J. Machado, Univ. Minho (PT)
Dragos Truscan, TUCS (FI)
João M. Fernandes, Univ. Minho (PT)

Program Committee:

- Jörg Desel, Katholische Univ. Eichstätt-Ingolstadt (DE)
- João M. Fernandes, Univ. Minho (PT)
- Marcus Fontoura, IBM (US)
- Luís Gomes, Univ. Nova Lisboa (PT)
- Jens B. Jorgensen, Univ. Aarhus (DK)
- Ridha Khedri, McMaster Univ. (CA)
- Bernd Kleinjohann, C-Lab (DE)
- Kai Koskimies, Tampere UT (FI)
- Maciej Koutny, Univ. Newcastle (UK)
- Johan Lilius, TUCS (FI)
- Ricardo J. Machado, Univ. Minho (PT)
- Ana Moreira, Univ. Nova Lisboa (PT)
- Ian Oliver, Nokia (FI)
- Carlos E. Pereira, UFRGS (BR)
- Ivan Porres, TUCS (FI)
- João P. Sousa, CMU (US)

Organizing Committee:

- Johan Lilius, TUCS (FI)
- Ricardo J. Machado, Univ. Minho (PT)
- Dragos Truscan, TUCS (FI)
- João M. Fernandes, Univ. Minho (PT)

Local Arrangements:

- Eric Badouel, INRIA/IRISA (FR)
- Laure Petrucci, Univ. Paris XIII (FR)

Contents

Modeling and Veri cation of Cryptographic Protocols Using Coloured Petri Nets and *Design/CPN*

Issam Al-Azzoni, Douglas G. Down, and Ridha Khedri
McMaster University
1280 Main Street West, Hamilton, Ontario, Canada L8S 4K1
{alazzoi, downd, khedri}@mcmaster.ca

Abstract

In this paper, we present a technique to model and analyse cryptographic protocols using coloured Petri nets. A model of the protocol is constructed in a top-down manner: rst the protocol is modeled without an intruder, then a generic intruder model is added. The technique is illustrated on the TMN protocol, with several mechanisms introduced to reduce the size of the occurrence graph. A smaller occurrence graph facilitates deducing whether particular security goals are met.
Keywords: Cryptographic protocols, Protocol analysis, Coloured Petri nets, Design CPN, Security goals.

1 Introduction

Cryptographic protocols play a crucial role in achieving security in today's communication systems. They are used in the Internet and in wired and wireless networks to ensure privacy, integrity and authentication. A cryptographic protocol is a communication protocol that uses cryptographic algorithms (encryption and decryption) to achieve certain security goals.

Generally, a cryptographic protocol involves two communicating agents who exchange a few messages, with the help of a trusted server. The exchanged messages are composed from components such as keys, random numbers, timestamps, and signatures [13]. At the end of the protocol, the agents involved may deduce certain properties such as the secrecy and authenticity of an exchanged message [12].

In analysing a cryptographic protocol, all possible actions by an intruder must be considered. An intruder is an attacker who wants to undermine the security of a protocol. An intruder can perform the following actions to mount attacks [12]: prevent a message from being delivered, make a copy of messages, intercept a message by preventing it from reaching its destination and making a copy, fake a message, modify a message, replay a message, delay the delivery of a message, and reorder messages. A fake message is fully generated using material gleaned from past exchanged messages while a modi ed message is a genuine message that the intruder partially altered.

The intruder manipulates messages as outlined above to mount an attack on the protocol. In this paper, we are concerned with attacks that result from a ws inherent in the protocol. Flaws in cryptographic protocols may allow an intruder to authenticate as someone else, or gain information that should not be otherwise revealed. We assume cryptographic algorithms are secure; *i.e.* it is not possible to decrypt a ciphertext without knowledge of the decryption key. This assumption allows us to focus on nding a ws inherent in the analysed protocol structure.

In this paper, we explore the use of coloured Petri nets [5] in the veri cation of cryptographic protocols. The ability to model concurrent behaviour has made coloured Petri nets an appropriate analysis tool for cryptographic protocols. There are two distinctive advantages of using coloured Petri nets: they provide a graphical presentation of the protocol, and they have a small number of primitives making them easy to learn and use. Furthermore, there exists a large variety of algorithms for the analysis of coloured Petri nets. Several computer tools aid in this process.

The computer tool *Design/CPN* [4, 10] had not been explored as a potential automated veri cation tool. We claim that given the power of *Design/CPN*, one can construct a coloured Petri net model of a cryptographic protocol and use advanced features to allow stronger and more ef cient veri cation. Examples of such features include: inscriptions, occurrence graph tools, hierarchical features, and ML queries.

In this paper, we are motivated to explore the use of Jensen's form of coloured Petri nets and *Design/CPN* in the veri cation of cryptographic protocols. In the process, we develop a new technique that addresses limitations of the techniques developed in [2, 14]. We focus on bene ting from the high level constructs of Jensen's coloured Petri nets, as well as using *Design/CPN*.

In the next section, we give an outline of the new technique. In Section 3, we demonstrate the technique by using it in the modeling and analysis of the TMN protocol. Finally, Section 4 summarises the technique's bene ts and suggests possible extensions. An extended version of this work can be found in [1].

2 Outline of the Technique

Our technique is a nite-state analysis method. Thus, it involves modeling the protocol as a coloured Petri net, then an automated tool (*Design/CPN*) is used to generate all possible states. Insecurities are discovered if an insecure state is reachable in the CPN occurrence graph.

The technique has several technical features not existing in other cryptographic protocol veri cation techniques using Petri nets. One of these features is the use of a central place to hold the tokens intercepted by the intruder; we call this place a DB-place. Its marking models the accumulated intruder knowledge. It is implemented by using a global fusion set of places. Although the pages of the illustrative example presented here are not big enough to fully illustrate the advantages of the use of fusion places, their use is extremely advantageous when one deals with more complex protocols. The colour set of this fusion set is de ned to be the union of the colour sets of tokens that can be possessed by the intruder. The use of the DB-place makes the intruder model simple and clear.

We implement a token-passing scheme to prevent unnecessary interleaving of the rings of protocol entity transitions. This results in a smaller occurrence graph. Other techniques [2, 14] handle the issue of state explosion differently. They restrict the behaviour of the intruder by introducing new assumptions. On the other hand, the intruder model in our technique is less restricted. This implies that our technique may capture a larger variety of attacks.

We use a top-down modeling approach. At the highest level of abstraction, an entity is modeled as a substitution transition. Each substitution transition is de ned in a separate subpage that provides a lower level description of the behaviour of the entity.

In modeling a cryptographic protocol using our technique, we follow these steps:

1. Build a model with no intruder: In this step a) using CPN ML notation, we declare the colour sets, functions, variables, and constants that will be used in the net inscriptions of the CPN model; b) we build a top-level model in which the protocol entities are modeled as substitution transitions; c) we de ne the substitution transitions from the top-level model.

2. Add the intruder to the model: In this step, a) we extend the CPN declarations to include the intruder; b) add the intruder transition to the top-level model; c) de ne the intruder substitution transition.

3. Implement a token-passing scheme.

4. Specify security requirements stated in terms of CPN markings.

5. Analyse the resulting occurrence graph by using OG queries to locate markings that violate a security requirement.

3 The Technique

In this section, we rst present our sample protocol, TMN protocol, and then, using our technique, we propose a model of this protocol. The selection of the TMN protocol to illustrate our technique is motivated by its familiarity. Any other protocol listed in [7] can be used for the illustration of our technique.

3.1 The TMN Protocol

The Tatebayashi, Matsuzaki, and Newman (TMN) protocol is a key exchange cryptographic protocol for mobile communication systems. The protocol involves two entities, A and B, and a server, J, to facilitate the distribution of a session key, K_{AB}. The attack illustrated in this paper is a known one. The reader can nd very similar attacks in [7, 8]. Moreover, three other known attacks on this protocol are given in [7].

Initially, the TMN protocol assumes that both A and B know the public key of J, K_J^{Pb}. We use the following notation: $(i) \quad A \rightarrow B : X$ to indicate that in the ith step of the protocol agent A sends message X to agent B. We write $A \rightarrow B : X, Y$ to denote "A sends B the message X along with the message Y". Furthermore, $key(data)$ denotes the message $data$ encrypted using the key key. The protocol proceeds as follows:

$$\begin{array}{ll} (1)\ A \rightarrow J : B, K_J^{\text{Pb}}(K_{AJ}) \qquad & (3)\ B \rightarrow J : A, K_J^{\text{Pb}}(K_{AB}) \\ (2)\ J \rightarrow B : A & (4)\ J \rightarrow A : B, K_{AJ}(K_{AB}) \end{array}$$

When A (the initiator) wants to start a session with B (the responder), A chooses a key K_{AJ}, encrypts it using the public key of J (K_J^{Pb}), and sends it along with the identity of B to the server J (step 1). Upon receiving the rst message, the server decrypts $K_J^{\text{Pb}}(K_{AJ})$ using its private key (K_J^{Pr}) and obtains K_{AJ}. Then, in the second step, J sends a message to B containing the identity of A. When B receives this message, it chooses a session key K_{AB}, encrypts it using K_J^{Pb}, and sends it along with the identity of A to J (step 3). Upon receiving the third message, the server decrypts $K_J^{\text{Pb}}(K_{AB})$ using its private key (K_J^{Pr}) and obtains K_{AB}. Then, J sends to A the key K_{AB} encrypted under K_{AJ} along with the identity of B (step 4). When A receives this message, it decrypts it using the key K_{AJ}, to obtain the session key K_{AB}.

The keys K_{AJ} and K_{AB} are symmetric keys freshly created by A and B, respectively. The key K_{AJ} must be known only to A and J; and is used to send K_{AB} in an encrypted form as indicated in step 4 of the protocol. The key K_{AB} must be known only to A, B and J, and it is used as a session key. Thus, A uses K_{AB} to encrypt messages it sends to B, and vice versa. When the communication session between A and B is over, K_{AB} is discarded. A new session key is used in every protocol run.

3.2 Modeling the Protocol with no Intruder

CPN ML Declarations

In our modeling of cryptographic protocols, messages are composed of elds. Some of these elds are atomic, they include entity identities, keys, and nonces. The other elds are constructed from the atomic elds. For instance, a cipher $K(A)$ can be viewed as an ordered pair (A, K), where A is the identity and K is the encryption key.

For the TMN protocol, we de ne the following:

1. Colour sets:

a) The atomic elds are the set of identities, $I = \{A, B\}$, and the set of keys, $K = \{K_{AB}, K_{AJ}, K_J^{\text{Pb}}, K_J^{\text{Pr}}\}$.

b) All ciphers have the same format: $k1(k2)$, where $k1$ and $k2$ are of type K. For instance, $K_J^{\text{Pb}}(K_{AJ})$ is the cipher of the rst message, *etc*. Thus, the cipher colour set C is de ned as $C = K \times K$.

c) Messages are generally composed of an identity and a cipher. For instance, in the TMN protocol, the rst message is $(B, K_J^{\text{Pb}}(K_{AJ}))$ and the third message is $(A, K_J^{\text{Pb}}(K_{AB}))$. Thus, the message colour set M is de ned as $M = I \times C$. Note that the second message of the protocol only includes an identity. Hence, I is used as the colour set for such messages.

d) The TMN protocol implicitly assumes that J knows the originator of the rst message it receives. To model this, we de ne the colour set $MI = M \times I$. Thus, the rst message that J receives is actually composed of two elds: the message contents, $(B, K_J^{\text{Pb}}(K_{AJ}))$, and the sender's identity A.

e) We use a special colour set $E = \{e\}$ to prevent an in nite number of transition rings. For instance, by using a construct such as in Figure 1, we force the transition T to re at most once. Using such constructs is needed whenever a transition has double input arcs. As we show later, the labels of double arcs indicate tokens inherent to an entity, or tokens that are to be used in subsequent subtasks performed by the entity.

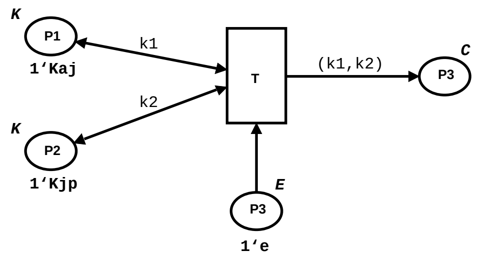


Figure 1: By using the E set, transition T can re at most one time.

2. Variables: We use variables of the de ned colour sets as inscriptions for arcs of the CPN. They are: $k1$, $k2$, and k of type K, c of type C, i of type I, and m of type M.

3. Functions:

a) The function $DecryptionKey(k : K)$ returns the decryption key of a given key k.

b) The function $SharedKey(i : I)$ returns the shared key between entity B and the entity i. For instance, $SharedKey(A)$ is K_{AB}. We use this function to model the behaviour of B in which it generates a session key based on the initiator's identity, as shown in step 2 of the protocol.

The TMN protocol declarations are given in Figure 2 using *CPN ML* notation.

```
color I = with A | B;
color K = with Kaj | Kjp | Kjpr | Kab;
color C = product K*K;
color M = product I*C;
color MI = product M*I;
color E = with e;
var k1,k2,k:K;
var c:C;
var i:I;
var m:M;
fun DecryptionKey(k:K):K = case k of Kaj=> Kaj
| Kjp => Kjpr | Kjpr => Kjp;
fun SharedKey(i:I):K= case i of A => Kab;
```

Figure 2: The declarations for the TMN model

The Top-Level Model

The computer tool *Design/CPN* supports hierarchical net construction. This makes it possible to model cryptographic protocols in a modular way. Thus, the model of a protocol is constructed by using sub-models of its agents. In CP-nets, this is implemented by using substitution transitions.

First, we focus on the messages exchanged between the protocol entities. At this level, protocol entities are modeled as transitions. Figure 3 shows a top-level model of the TMN protocol. This net is described as follows:

1. Transition $T1$ represents entity A. In the rst step of the protocol, A generates a token of type MI. This corresponds to the rst message that A sends to J, along with the identity of A to inform J about the initiator. In the last step of the protocol, A consumes a token of type M.

2. Transition $T2$ represents entity J. In the rst step of the protocol, J consumes a token of type MI. Then, J sends a token of type I, modeling the second protocol step. In the third step of the protocol, J consumes a token of type M generated by B. Finally, J generates a token of type M modeling the last message of the protocol.

3. Transition $T3$ represents entity B. Entity B consumes a token of type I in the second step, and generates a token of type M in the third step of the protocol.

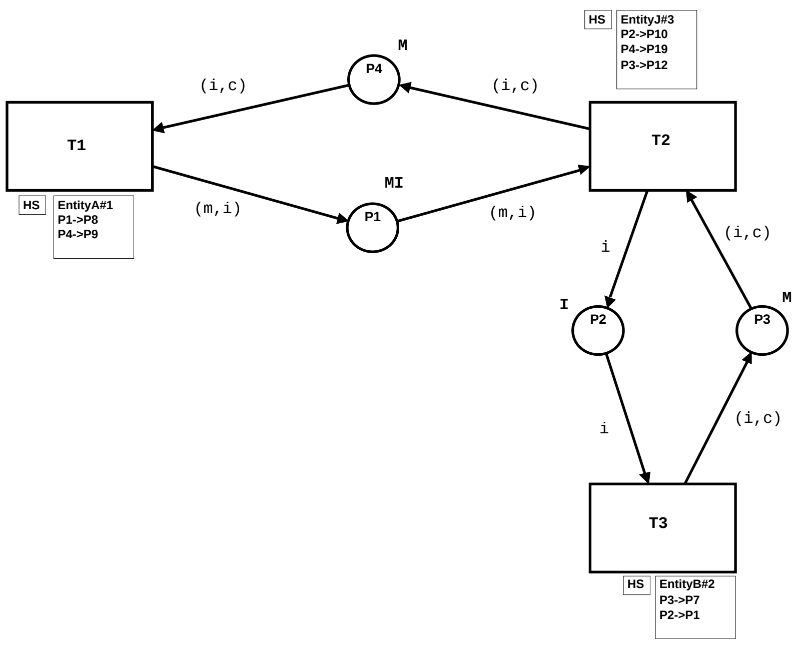


Figure 3: The TMN top-level model with no intruder

De ning the Top-Level Substitution Transitions

Due to space constraints, we consider in detail the model of the initiator A but we refer the reader to [1] for the models of entities B and J. The following is an informal description for the behaviour of A which is the initiator of the communication session. Thus, A always sends the message $B, K_J^{\mathrm{Pb}}(K_{AJ})$. The reply A receives is in the format (i, c), where i is an identity and c is a cipher. Entity A checks that $i = B$. If this is true, A decrypts c with K_{AJ}. If c was decrypted with K_{AJ}, A accepts the received session key, and uses it for communication with B in the current session.

Figure 4 shows the CPN model of entity A. It contains two subnets: one models the subtask of A initiating a protocol run in step 1, while the second models the subtask of A receiving the last message from J.

Port assignments are used to relate the top-level page, named *TMN*, with the entity models. As the port assignments for the substitution transition of A show (Figure 3), the socket $P1$ is related to the output port $P8$ of $EntityA$, while the socket $P4$ is related to the input port $P9$ of $EntityA$.

In $EntityA$, we use the instance fusion sets $B = \{P1, P10\}$ and $Kaj = \{P2, P12\}$. Fusion sets are used to allow an entity to control the order of subtasks and check the validity of messages. For instance, A has to remember the key it chooses (K_{AJ}) in the rst step, in order to decrypt the cipher it receives in the last step.

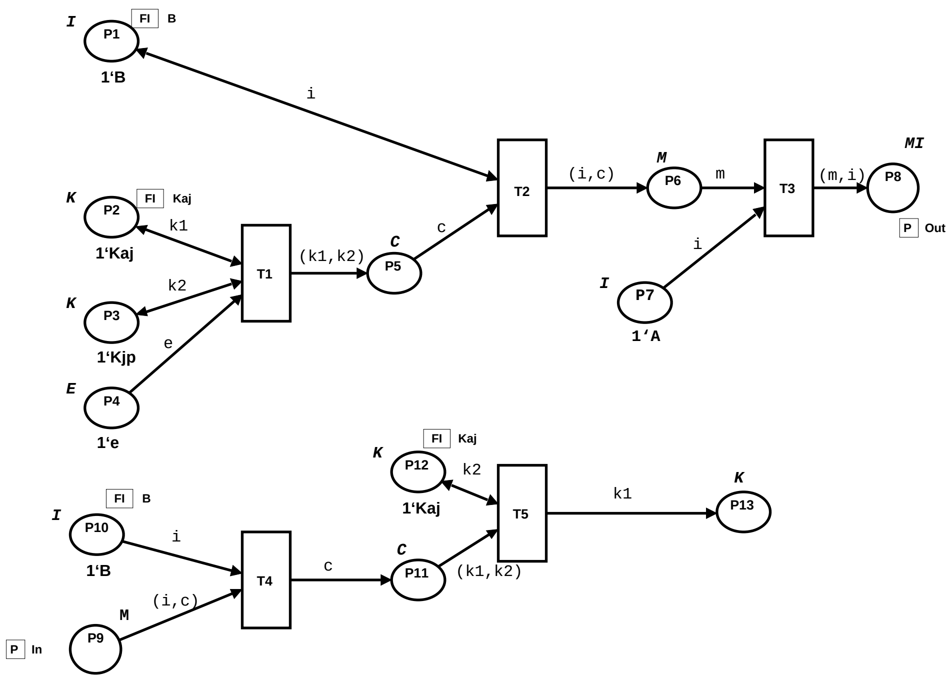


Figure 4: Page $EntityA$

3.3 Modeling the Protocol with an Intruder

The intruder is modeled as a separate entity that controls the communication channels between the protocol entities. Thus, it intercepts the exchanged messages and stores them for future use. Then, it attempts to decrypt the encrypted portions of the intercepted messages. Finally, it attempts to modify the message contents, or even generate new messages to replace the intercepted ones.

Extending the CPN ML Declarations

In order to add the intruder to the model, one must extend the CPN ML declarations. The identity of the intruder In is added to the colour set I. Also, an intruder key Ki is added to the colour set K. The $DecryptionKey$ and $SharedKey$ functions are extended to handle the new colours: $DecryptionKey(Ki) = Ki$ and $SharedKey(In) = Ki$.

During the execution of the protocol, the intruder stores the intercepted messages for future use. We model the intruder memory as a global fusion set that we call the DB fusion set (DB stands for database). We refer to a place that is a member of the DB fusion set as a *DB-place*.

A DB-place is expected to hold tokens of atomic and non-atomic types. In the TMN protocol, a DB-place should hold keys, identities, and ciphers. Thus, we de ne the DB colour set as DB $= I \cup K \cup C$, and we use DB as the colour set of a DB-place.

In CPN ML, DB is declared as follows: *color DB=union cI:I + cK:K + cC:C;* Here, *cI*, *cK*, and *cC* are selectors [10]. Thus, the intruder's possession of K_{AB} is modeled as reaching a marking where a token $cK(Kab)$ is in a DB-place. The reader can nd the nal CPN ML declarations in [1].

The Top-Level Model with an Intruder

Figure 5 shows the top level model of the TMN protocol with an intruder. The substitution transition $T4$ represents the intruder, which was not included in the earlier top level model given in Figure 3.

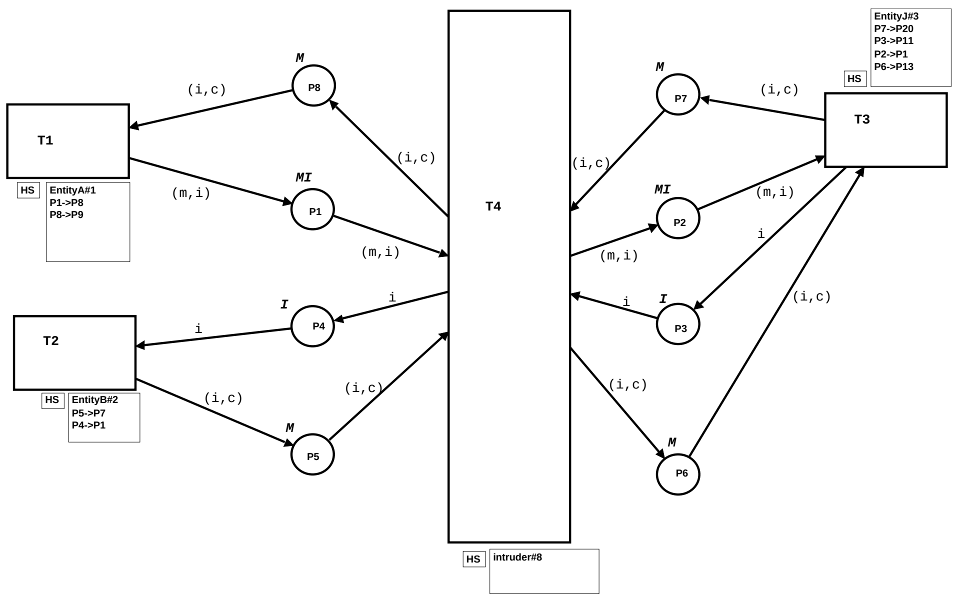


Figure 5: The TMN top-level model with an intruder

Each place in Figure 3 is replaced with two corresponding places as shown in Figure 5: one is an input place to the intruder while the second is an output place. For instance, place $P1$ in Figure 3 is replaced with $P1$ and $P2$ in Figure 5. This is needed to model the intruder's ability to receive a message (the input place), to deal with it (transition $T4$), and to substitute it with a new message (the output place).

De ning the Intruder Substitution Transition

The intruder substitution transition ($T4$ in Figure 5) is de ned by the subpage $intruder$ shown in Figure 6.

The intruder model is constructed by using several intruder subprocesses. Each intruder subprocess models the intruder's possible actions to intercept tokens

Table 1: The intruder subprocesses

Pair Places		Colour	The Corresponding
Input	Output	Set	Intruder Subprocess
P1	P2	MI	intruder_mi
P3	P4	I	intruder_i
P5	P6	M	intruder_m
P7	P8	M	intruder_m

that belong to a given colour set (type). Table 1 lists the intruder subprocesses, along with their input/output places.

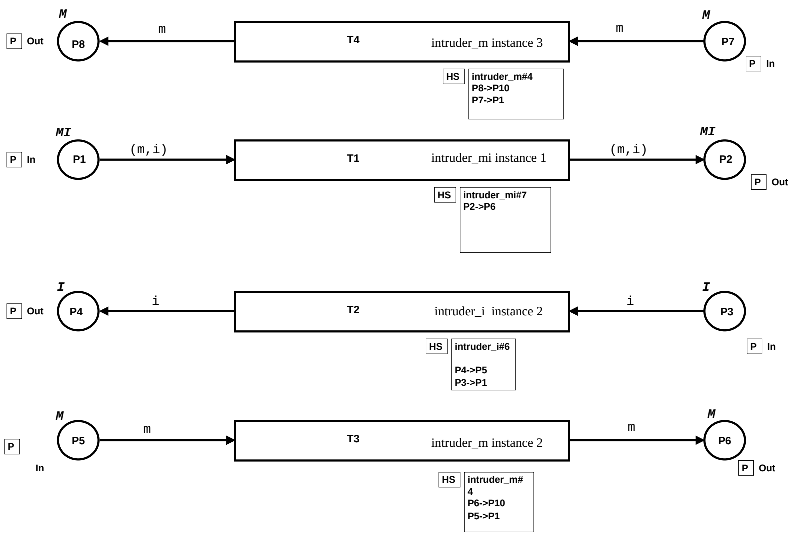


Figure 6: The *intruder* page

The intruder subprocesses $intruder\_mi$, $intruder\_m$, and $intruder\_i$ are de ned in separate pages. The $intruder$ page has one instance of $intruder\_mi$ (which de nes $T1$), one instance of $intruder\_i$ (which de nes $T2$), and two instances of $intruder\_m$ (which de ne $T3$ and $T4$).

The $intruder\_m$ subprocess is given in Figure 7. It models what an intruder can do to intercepted tokens of type M. A token of type M has two elds: an identity and a cipher. The intruder rst stores these elds of the intercepted token. Then, it tries to decrypt the cipher using one of the keys stored in its database. Finally, the intruder forms a new message to be sent in place of the intercepted one. The intruder uses one of the ciphers stored in the database, or constructs a new cipher by using keys stored in the database.

The $intruder\_i$ subprocess models what an intruder can do to intercepted

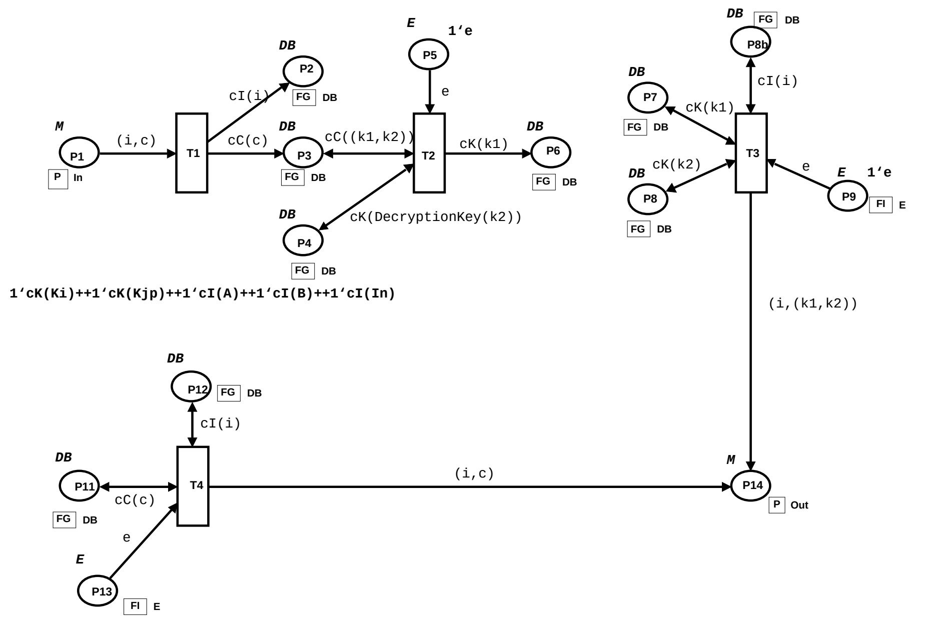


Figure 7: Page $intruder\_m$

tokens of type I. It is constructed in a similar manner as $intruder\_m$ (for details, see [1]). The $intruder\_mi$ subprocess models what an intruder can do to intercepted tokens of type MI. A token of type MI has two elds: an identity and a message. Thus, $intruder\_mi$ can be constructed using instances of $intruder\_i$ and $intruder\_m$, as shown in Figure 8. An instance of $intruder\_i$ is used to handle the identity eld, and an instance of $intruder\_m$ is used to handle the message eld.

The last step in de ning the intruder is to specify its initial knowledge. One speci es the initial intruder knowledge by setting the initial marking of a DB-place. As the initial marking of $P4$ in $intruder\_m$ indicates (Figure 7), the DB is set initially to $\{K_I, K_J^{\text{Pb}}, A, B, In\}$.

3.4 Applying a Token-Passing Scheme

Using our technique as outlined up to this point, most models of cryptographic protocols result in a large occurrence graph. The large size of the occurrence graph can be explained by two aspects of the model: the nondeterministic behaviour of the intruder, and the interleaved subprocesses.

The intruder model is nondeterministic in the sense that there are many possible actions the intruder can take at a given time. For instance, assume that in the TMN model the intruder has the keys K_I and K_J^{+}, and it has three identities: A, B, and I. Then, there are 12 possible messages (i, c) the intruder can use. Each

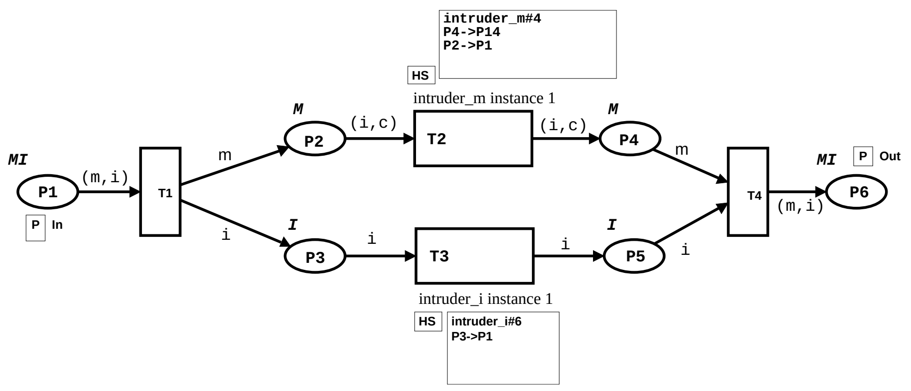


Figure 8: Page $intruder\_mi$

choice will have different implications in terms of the resulting markings.

The second factor attributing to the size of the occurrence graph is the interleaving of subprocesses. Transitions of an entity and the intruder instances can be interleaved, causing an unnecessary increase in the size of the occurrence graph. For instance, consider a state where transition $T1$ of $EntityA$ has not yet red. At this state, many transitions of the intruder instances are enabled. The different order of ring such transitions will result in different markings and paths in the occurrence graph. The same thing happens after ring $T2$ of *EntityA*, etc.

Let $\mathcal{E}$ be the set of nite occurrence sequences of the possible execution of the agents A, B, and J. For every sequence e, the intruder observes the set S_e of relevant information (keys, messages, and agent identities) that are carried by e. Let $R = \{(e_1, e_2) \mid e_1 \in \mathcal{E} \wedge e_2 \in \mathcal{E} \wedge S_{e_1} = S_{e_2}\}$. The relation R is an equivalence relation. It is clear that interleaving of subprocesses belong to the same R-equivalence class as their sequential execution. Hence, there is no need to include all the the unnecessary interleaving of subprocesses in the occurrence graph.

The model can be extended to prevent the unnecessary interleaving of subprocesses. The goal is to allow a single subprocess to be enabled at a given time. This is achieved using a token-passing scheme. For instance, if *EntityA* has the token, no transitions from other subprocesses should re. This results in a reduction in the size of the occurrence graph.

We note that applying the token-passing scheme does not restrict the model assumptions. This is because it is assumed that an intruder would not obtain more knowledge by the simultaneous execution of protocol entities than it would by the interleaving of such executions. In other words, true concurrency is assumed not to affect properties of cryptographic protocols.

To apply the token passing strategy, a new colour set is de ned, $S = \{s\}$. We will refer to a place of colour S as an *S-place*. The token s is the token exchanged among entities.

The following rules are the changes required to apply this scheme.

1. Add an input S-place to every substitution transition in the top level page. Similarly, add an output S-place from every substitution transition in the top level page. All of these S-places should be added to a single instance fusion set. Thus, there is one resulting S-place. It must be initialised with one s-token. This rule is demonstrated in Figure 9.

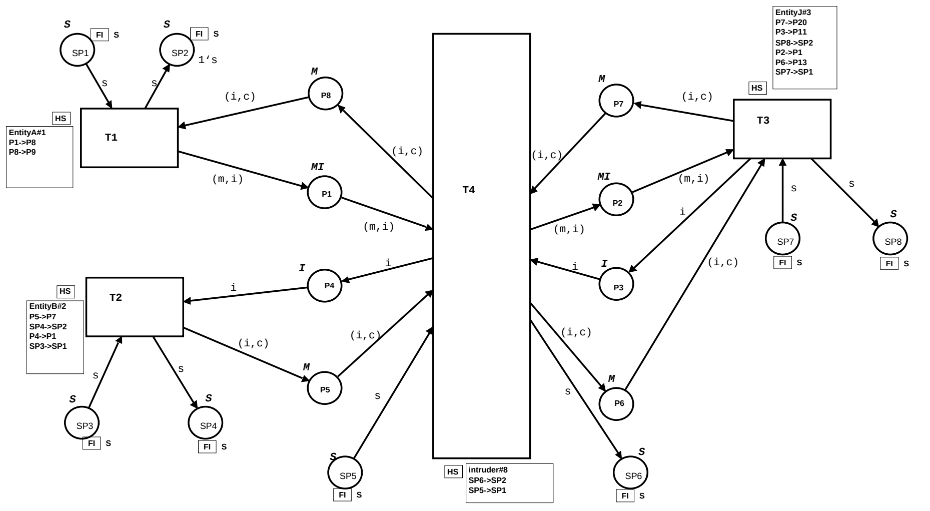


Figure 9: The *TMN* page after adding the S-places

2. Add an S-place input port and an S-place output port to every subpage. The input port should have an outgoing arc to the rst transition in every subprocess of the subpage. Similarly, the output port should have an incoming arc from the last transition in every subprocess of the subpage. Figure 10 shows the application of this rule to the $EntityA$ page.

3. Applying the rst two rules does not prevent the intermediate intruder transitions from ring. These are the transitions that have double input arcs coming from DB-places, *e.g.* transitions $T2$ and $T3$ in $intruder\_m$. We must allow these transitions to re only when the corresponding subprocess has the s-token. To apply this, we create an instance fusion set S, in every intruder subpage, to hold the s-token that is passed to the active intruder subprocess. To be more precise, a) we add an output arc from the rst transition of the intruder subpage to a place that belongs to the fusion set S; b) we add an input arc from a place that belongs to the fusion set S to the last transition of the intruder subpage; and c) we add double arcs from a place that belongs to the fusion set S to the intermediate intruder transitions.

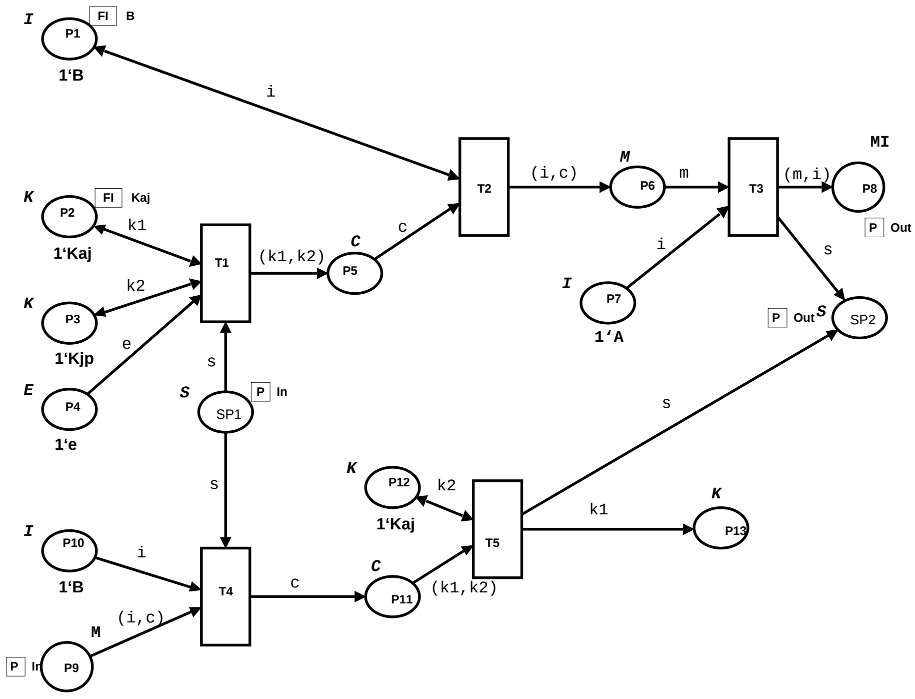


Figure 10: The $EntityA$ page after adding the S-places

These changes are demonstrated in Figure 12. For example, transitions $T2$ and $T3$ of $intruder\_m$ will not re until the s-token arrives to the subprocess, which means transition $T1$ res, consuming the s-token from the input port $SP1$. When the s-token is returned back by the intruder subprocess (*i.e.* transition $T4$ res and the s-token is deposited back to the output port $SP2$), transitions $T2$ and $T3$ become disabled.

Note that the intruder intermediate subpages, *e.g.* $intruder\_mi$, must be extended to pass the received token to the lower level subpages. This is demonstrated in Figure 11.

The application of these rules to the pages $EntityB$, $EntityJ$, $intruder$ and $intruder\_i$ is provided in [1].

3.5 Identifying Security Requirements

Before simulating the model, one needs to identify the security requirements that must be met by the protocol. These requirements should be stated in terms of conditions on the CPN markings.

We consider the following requirement. The protocol must guarantee the secrecy of the session key K_{AB}. Thus, in a given session, K_{AB} must be known only by A,B, and J. In other words, the intruder should never know K_{AB}. In

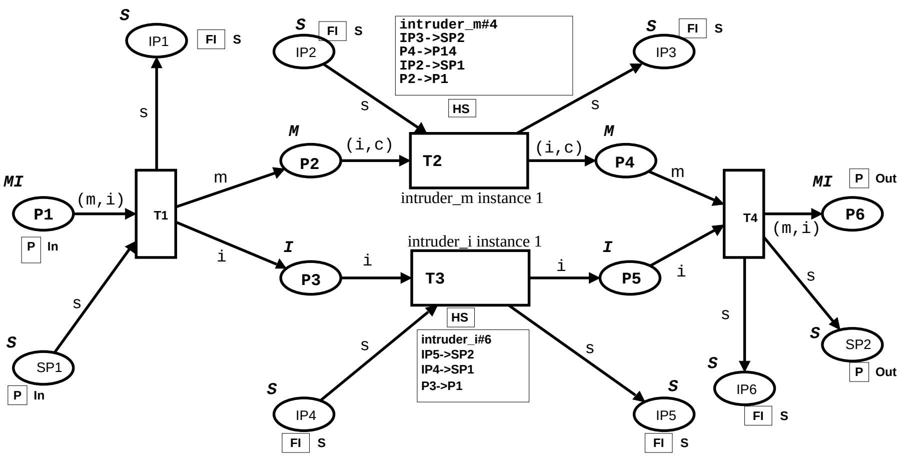


Figure 11: The $intruder\_mi$ page after adding the S-places

terms of CPN markings, this translates into the requirement that a token with colour Kab never reaches a DB-place.

Other security requirements that the TMN protocol aims to satisfy are discussed and veri ed in [1].

3.6 Analysing the Occurrence Graph

The nal step in the analysis of the model is to construct and analyse the occurrence graph. We use the OG tool in *Design/CPN* to automate this process. The goal is to nd nodes (markings) that violate a security requirement.

We use the *Occ Menu* to invoke commands related to the occurrence graph [11]. Given the CPN model for a cryptographic protocol, we construct the full occurrence graph, and then run CPN queries to nd the insecure markings.

The security requirement that we consider states that a token with colour Kab never reaches a DB-place. In CPN ML, we use the following predicate: `fn n => cf(cK(Kab), Mark.intruder_m'P4 1 n) >0`. Given a marking n, this predicate evaluates to true if the DB-place $P4$ of $intruder\_m$ (rst instance) contains at least one token $cK(Kab)$, and evaluates to false otherwise. Note that cf is the coef cient function [10]. It takes two arguments: a colour and a multiset of tokens, and returns the coef cient of the speci ed colour in the speci ed multiset. For instance, `cf(A,5'A)` returns 5. Thus, `cf(cK(Kab), Mark.intruder_m'P4 1 n)` returns the coef cient of $cK(Kab)$ in the multiset of tokens in $P4$ of the rst instance of $intruder\_m$ in marking n.

The following function returns all nodes of the occurrence graph where the DB-place has at least one token $cK(k)$. It uses the predicate de ned above.

```
    fun SecrecyViolation1(k:K):
```

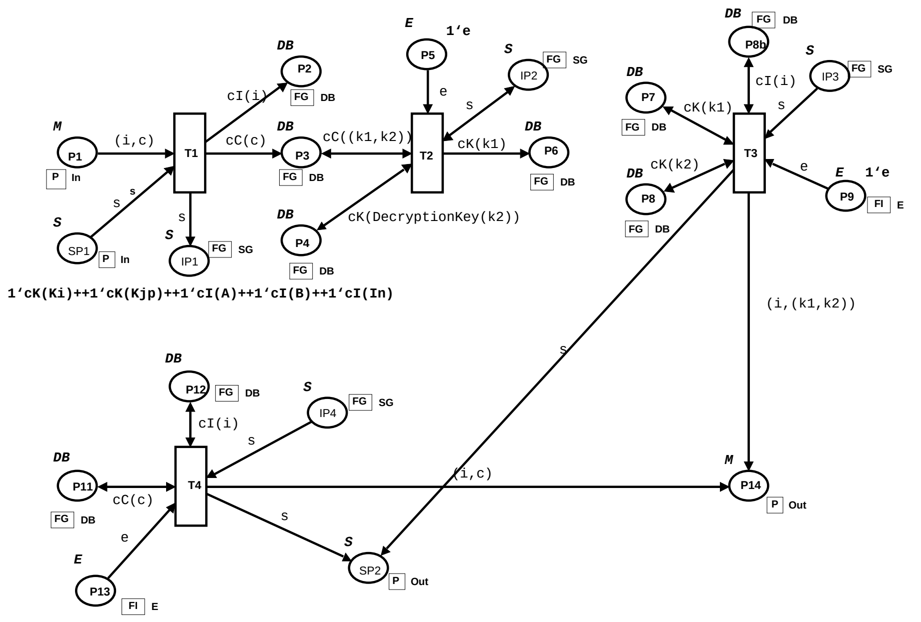


Figure 12: The $intruder\_m$ page after adding S-places

```
Node list
= PredAllNodes (fn n => cf(cK(k),
Mark.intruder_m'P4 1 n) >0);
```

Thus, $SecrecyViolation1(Kab)$ returns all nodes of the occurrence graph that violate the considered security requirement.

The full occurrence graph generated for the model has 19,237 nodes and 22,419 arcs. It took 19 seconds to construct the occurrence graph using a 1-GHz, 16GB machine.

Executing $SecrecyViolation1$ returns a non empty node list. One of the nodes returned by $SecrecyViolation1$ is node 19170. We use the *Design/CPN* Occurrence Graph (OG) tool to nd a path from the initial marking (node 1 in the OG) to the insecure marking (node 19170). This path is represented by the following occurrence sequence. Each line in the occurrence sequence represents a step that has a single binding element. Each line contains the following information: the page name, the instance number (if missing, then there is a single instance), the transition, and the binding. For instance, the line identi ed by (*), on its right side, represents the step ($T2$ in the rst instance of $intruder\_m$, $\langle k1 = Kab, k2 = Ki\rangle$).

EntityA	T1	$k1 = Kaj, k2 = Kjp$
EntityA	T2	$i = B, c = (Kaj, Kjp)$
EntityA	T3	$i = A, m = (B, (Kaj, Kjp))$

intruder_mi	T1	$m = (B, (Kaj, Kjp)), i = A$
intruder_i 1	T1	$i = A$
intruder_i 1	T2	$i = A$
intruder_m 1	T1	$i = B, c = (Kaj, Kjp)$
intruder_i 2	T2	$i = A$
EntityB	T1	$i = A$
EntityB	T2	$i = A, k2 = Kjp$
EntityB	T3	$i = A, c = (Kab, Kjp)$
intruder_m 2	T1	$i = A, c = (Kab, Kjp)$
intruder_m 1	T3	$k1 = Ki, k2 = Kjp, i = A$
intruder_mi	T4	$i = A, m = (A, (Ki, Kjp))$
EntityJ	T1	$i = A, m = (A, (Ki, Kjp))$
EntityJ	T2	$i = A, c = (Ki, Kjp)$
EntityJ	T3	$k1 = Ki, k2 = Kjp$
EntityJ	T4	$i = A, k = Ki$
intruder_i 2	T1	$i = A$
intruder_m 2	T4	$i = A, c = (Kab, Kjp)$
EntityJ	T5	$i = A, c = (Kab, Kjp)$
EntityJ	T6	$k1 = Kab, k2 = Kjp$
EntityJ	T7	$k1 = Kab, k2 = Ki$
EntityJ	T8	$i = A, c = (Kab, Ki)$
intruder_m 3	T1	$i = A, c = (Kab, Ki)$
intruder_m 1	T2	$k1 = Kab, k2 = Ki$ (*)
intruder_m 2	T2	$k1 = Kab, k2 = Ki$
intruder_m 3	T3	$i = B, k1 = Ki, k2 = Kjp$
EntityA	T4	$i = B, c = (Ki, Kjp)$

Note the reachability of K_{ab} to a DB-place in the step identi ed by (*). This attack is stated in a high level description as follows, noting that $I(A)$ denotes I impersonating A. Thus, a step of the form "$I(A) \rightarrow B : X$" means that I poses as A and sends X to B, whereas a step of the form "$B \rightarrow I(A) : X$" means that I intercepts the message X; originally sent from B to A.

$(I1)\ A \rightarrow I(J) : B, K_J^{\text{Pb}}(K_{AJ})$

$(I2)\ I(J) \rightarrow B : A$

$(I3)\ B \rightarrow I(J) : A, K_J^{\text{Pb}}(K_{AB})$

$(II1)\ I(A) \rightarrow J : A, K_J^{\text{Pb}}(K_I)$

$(II2)\ J \rightarrow I(A) : A$

$(II3)\ I(A) \rightarrow J : A, K_J^{\text{Pb}}(K_{AB})$

$(II4)\ J \rightarrow I(A) : A, K_I(K_{AB})$

This attack involves two separate runs of the protocol; labelled I and II. At the end of $II4$, the intruder decrypts $K_I(K_{AB})$ to obtain K_{AB}. Thus, the intruder is able to impersonate A. Note the replay of $K_J^{\text{Pb}}(K_{AB})$ in step $II3$ from $I3$.

4 Discussion

In this paper we have presented a promising technique that uses coloured Petri nets for the veri cation of cryptographic protocols.

Our technique compares well with other nite-space methods [8, 9]. It includes the same veri cation assumptions. The same approach of reachability analysis is used. The generated number of states is acceptable compared with other methods. Furthermore, *Design/CPN* ts well to our technique, with several advantageous features such as the ability to control the construction of the occurrence graph and the ability to stop searching when certain criteria are met. In other terms, the capabilities of *Design/CPN* enable us to grasp the theoretical power of CP nets in practice for dealing with complex systems. The state explosion problem can be slightly managed using for instance a token-passing scheme, but not signi cantly reduced. In [3], the *sweep-line method* is introduced to reduce both the space and the time used during state space exploration. One avenue for future investigation would be to apply this method in the exploration of the state space of more complex protocols modeled using the technique proposed in this paper.

There are two features in our technique that facilitate the construction of the intruder model for cryptographic protocols. The rst feature is the use of a DB-place to hold all intercepted tokens. The second feature is that the intruder model is constructed by using several intruder subprocesses, where each intruder subprocess is de ned based on the colour of the intercepted token. For instance, if the intercepted token is an identity, then the intruder rst stores it and then it replays any other identity it possesses. If the intercepted token is a cipher, the intruder has the ability to try to decrypt the cipher and to form new ciphers. The net result of this is clarity and simplicity of the intruder model, and the ability to construct the intruder model in a systematic way.

Finally, the model presented for the TMN protocol involves a single instance of each entity. Thus, an attack that involves multiple instances of a given entity in multiple runs will not be captured under this restriction. Our model can easily be extended to include more than one instance of a given entity by adding tokens to the entity's E-places. However, this would result in a dramatic increase in the size of the occurrence graph. This problem also arises in other nite-state methods. In such cases, analytic methods are applied to avoid generating the full reachability tree. For the case of CP-nets, methods such as the matrix equation [5] seem to be useful. Other techniques to yield a reduced representation of the occurrence graph are applicable. These include the stubborn set method [6], and occurrence graphs with equivalence classes [5].

References

[1] I. Al-Azzoni. The verification of cryptographic protocols using coloured Petri nets. Master's thesis, McMaster University, Hamilton, Ontario,

Canada, 2004.

[2] A. Basyouni and S. Tavares. New approach to cryptographic protocol analysis using coloured Petri nets. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE'97)*, pages 334–337, St. John's, Newfoundland, May 1997.

[3] S. Christensen, L. M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *Proceedings of TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464. Springer-Verlag, 2001.

[4] CPN Group at the University of Aarhus. Design/CPN Online, 2004. http://www.daimi.au.dk/designCPN/.

[5] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 2: Analysis Methods. Springer-Verlag, 2nd edition, 1996.

[6] L. M. Kristensen and A. Valmari. Finding stubborn sets of coloured Petri nets without unfolding. *Lecture Notes In Computer Science*, 1420:104–123, 1998.

[7] Laboratoire Spéci cation et Véri cation. SPORE security protocols open repository. http://www.lsv.ens-cachan.fr/spore/table.html. (accessed December 16, 2004).

[8] G. Lowe and B. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering*, 23(10):659–669, October 1997.

[9] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, Feb 1996.

[10] Meta Software Corporation. *Design/CPN Reference Manual for X-Windows*, Version 2.0, 1993.

[11] Meta Software Corporation. *Design/CPN Occurrence Graph Manual*, Version 3.0, 1996.

[12] P. Ryan and S. Schneider. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2001.

[13] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley, 2nd edition, 1996.

[14] D. M. Stal, S. E. Tavares, and H. Meijer. Backward state analysis of cryptographic protocols using coloured Petri nets. In *Workshop on Selected Areas in Cryptography, SAC '94 Workshop Record*, pages 107–118, May 1994.

Model Transformations for an Elevator Controller: Coloured Petri Nets in Object-oriented Analysis and Design

João Paulo Barros[1,2] and Jens Bæk Jørgensen[3]
[1]Instituto Politécnico de Beja, Escola Sup. de Tecnologia e Gestão
Rua Afonso III, n. 1, 7800-050 Beja, Portugal
[2]Universidade Nova de Lisboa/UNINOVA, Portugal
[3]Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
e-mail: jpb@uninova.pt, jbj@daimi.au.dk

Abstract

Coloured Petri nets (CPNs) are useful to model the behaviour of systems. CPN models are expressive, executable, and scalable. Besides, CPN have a standard and implemented formal, precise semantics, which allows model verification. In this paper, we first demonstrate how a CPN model can be used to capture requirements for a considered example system, an elevator controller. Then, we show how the requirements-level CPN model is transformed into an object-oriented design-level CPN model, which is structurally and conceptually closer to object-oriented programming languages. The transformation reduces the gap between CPN models and the respective implementation, thus simplifying the implementation or code generation phase. Finally, we discuss the code generation from object-oriented CPN models.

1 Introduction

Petri nets are sometimes seen as a formal language whose application is not scalable to large systems. This perspective results from equating Petri nets to low-level Petri nets, especially Place/Transition nets [19]. Yet, there are numerous classes of Petri nets, all of them sharing a few fundamental characteristics, e.g. graphical representation, precise semantics, and duality of concepts [5]. In particular, *coloured Petri nets* (CPNs) [9] allow the creation of compact, expressive, executable, hierarchical, and readable models.

This paper shows how to apply CPNs in object-oriented design. More specifically, we show that CPN models can effectively be used to express requirements in a precise and executable way. The resulting executable model can be used to provide feedback in the analysis phase; it describes the system behaviour and reactions to the possible external events it receives. This behaviour and reactions should be caused by the software system we are going to develop. The software system itself is described by a class diagram and an object-oriented CPN model. The object-oriented CPN model is closer to an implementation of the software than the requirements-level CPN model. In this way CPN models are used to reduce the gap between user-level requirements and implementations.

The paper is structured as follows: Section 2 presents the use cases for the considered elevator controller; Section 3 shows a requirements-level CPN model for the elevator controller while informally introducing CPN's syntax and semantics; starting from the use case diagram; Section 4 presents a class diagram for the elevator controller; Section 5 presents a design-level object-oriented CPN model and Section 6 discusses code generation issues; finally, Section 7 discusses related work, and Section 8 concludes.

2 Use Cases

The elevator controller, we consider, must work in a ten floor building in which there are two elevator cages.

The main responsibility of the controller is to control the movement of the two cages. Movement is triggered by passengers, who push request buttons. On each floor, there are floor buttons, which can be pushed to call the elevator; a push indicates whether the passenger wants to travel up or down. Inside each cage, there are cage buttons, which can be pushed to request to be carried to a particular floor. In addition to controlling the movement of the cages, the controller is responsible for updating a location indicator inside each cage, which displays the current floor of the cage.

There are many use cases that must be supported by the elevator controller; examples are: (1) *Collect passengers*: When a passenger pushes a floor button on floor *f*, eventually an elevator cage should arrive at floor *f* and open its doors; (2) *Deliver passengers*: When a passenger pushes the cage button for floor f in an elevator cage, eventually the elevator cage should arrive at floor f and open its doors; (3) *Show floor*: When a cage arrives at a floor, passengers inside the cage should be informed about the current floor number. The relationship between the use cases and external entities in the environment of the elevator controller are depicted in Figure 1.

Let us take a closer look at the use case *Collect passengers*, whose trigger event is the push of a floor button. This should generate a stimulus to the elevator controller, which, upon reception, must do a number of things: (1) The controller must turn on the light of the button that was pushed; (2)

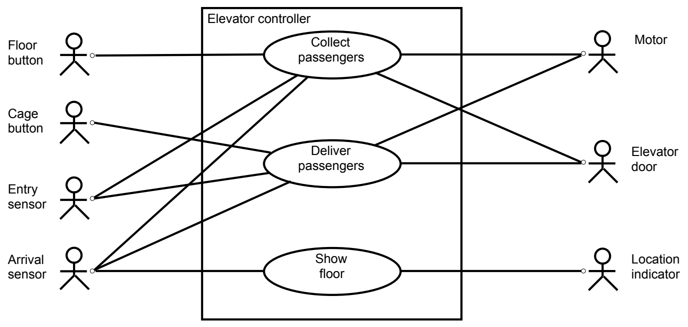


Figure 1: UML-style use case diagram for the elevator controller.

the controller must allocate the request to one of the cages. In particular, this implies that the controller must determine whether the request can be served immediately. This is possible only if the request comes from a floor where there currently is an idle cage. In this case, the cage can just open its doors; it is not necessary to start the motor; (3) if it is necessary to start the motor, the controller must generate an appropriate signal to the motor; (4) if it is sufficient to open the doors, the controller must generate a signal to the doors instructing them to open.

We could illustrate this scenario and its continuation with cages moving, sensors being triggered, etc. using, e.g., sequence diagrams. Alternatively, we could describe the desired general behaviour of the elevator controller and the entities in its environment using statecharts [7]. This is done, e.g., by Wieringa in [21]. However, our objective with this paper is to demonstrate the applicability of CPNs [9] in object-oriented analysis and design (statecharts and CPN models have their advantages and disadvantages in comparison with each other, see, e.g., [6, 12]). Therefore, instead, we will describe the desired general behaviour using CPN. We do so in the next section.

3 Requirements-level CPN Model

We have built a CPN model, which models the desired behaviour of the environment as controlled by the elevator controller.

The model is created and executed with the tool *CPN Tools* [3], which has a graphical part and includes the programming language Standard ML [16]. Together with the explanation of the model, this section is an informal primer to the CPN language itself, which allows the reader to understand the model in general terms — although we do not explain all the technicalities.

The CPN model consists of (1) *declarations* of data types, functions, etc. and (2) a *graphical net structure* in the form of three related modules: `Do Cage Cycle`, `Handle Requests`, and `Move UpDown`. As we will see, the declarations are used as inscriptions in the graphical net structure.

3.1 Representation of Entities in the Environment

Entities in the environment are represented via data type declarations. This applies both to the environment entities, which can be seen from the use case diagram in Figure 1, but also relevant entities with which the controller does not interact directly, but which it controls via other entities — e.g., the controller controls a cage because the controller interacts with the cage's motor.

As an example, the data type `CAGE` used to represent the elevator cages consists of 4-tuples of the form `(cageid,floor,requestlist,direction)`; `cageid` identifies the cage; `floor` is the number of the floor the cage currently is at, if the cage is stationary, or has last visited, if the cage is moving; `requestlist` is the cage's request list represented as a list of floor numbers; `direction` holds the cage's current direction of movement: `up`, `down`, or `no`.

Other examples are floors that are represented as integers, and a floor button as a pair `(floor,direction)`, where `direction` is `no` if the button has not been pushed and otherwise `up` or `down`, indicating the passenger's direction request. The cage buttons in each cage are represented as a pair `(cageid,buttonlist)`, where `buttonlist` is a list of integers corresponding to the floors for which cage buttons currently have been pushed.

3.2 Representation of Cage Behaviour

The behaviour of cages is modelled in the `Do Cage Cycle` module, shown in Figure 2.

A CPN model describes both states and events. The state of a CPN model is a distribution of *tokens* on *places*. The latter are drawn as ellipses. Each place has a data type, written in capital letters, which determines the kinds of tokens the place may contain. In Figure 2, the two elevator cages are modelled by `CAGE` tokens. Each `CAGE` token is at any time on exactly one of the places `Idle`, `Moving`, `Opened`, or `Closed`, which all have data type `CAGE`. The `Floor Buttons` and `Cage Buttons` places are used to model the floor buttons and cage buttons, respectively. Tokens on these places correspond to buttons and model whether buttons are on or off. The state shown in Figure 2 is the model's initial state, which represents a situation, where both elevator cages are idle on floor 1 and no requests have been made.

The events of a CPN model are represented using *transitions*, drawn as rectangles. *Arcs* connect transitions with places. The events consist of

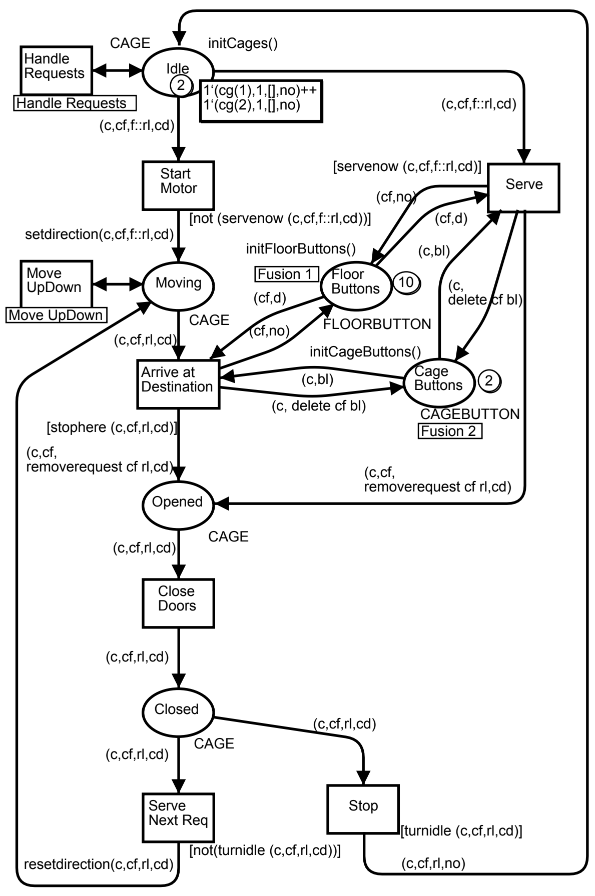


Figure 2: Do Cage Cycle module of the requirements-level CPN model.

transitions that remove tokens from input places and add tokens to output places. A transition that is ready to remove and add tokens is *enabled*; it may *occur*. There are two conditions for enabling: (1) Appropriate tokens are present on the input places — the values in these tokens are bound to the variables appearing in the inscriptions around the transition; (2) A guard — a Boolean expression in square brackets — is true.

As an example, enabling of `Start Motor` requires: (1) that `Idle` contains some `CAGE` token that matches the pattern `(c,cf,f::rl,cd)`, i.e., a `CAGE` token with a non-empty request list; and (2) that the guard `[not (servenow (c,cf,f::rl,cd))]` evaluates to true; i.e., that the state represents a situation in which cage `c` is not currently at floor `f`. `Start Motor` is not enabled in the shown state; `f::rl` is a non-empty list and there are no pending requests.

`Handle Requests` and `Move UpDown` in Figure 2 are special kinds of transitions that refer to the two other modules of the model, which we do not have space to describe in detail in this paper. The `Handle Requests` module describes the handling of requests, i.e., the making and subsequent allocation of requests to cages. The `Move UpDown` module describes the up and down movement of the elevator cages between the floors, and the update of the location indicators.

3.3 Use of the Requirements-level CPN Model

The CPN model we have just presented is described in more detail in [10]. The model can be seen as an *executable use case* in the sense of [11]. Execution of the model can be used to validate the three use cases of Figure 1. More generally, the CPN model can be used as a vehicle for requirements engineering. The model and its execution can be used to specify, validate, and elicit requirements for the elevator controller.

The model describes when the controller should interact with external entities like motors, buttons, sensors, and doors. The model also describes what should be the effect of such interactions in the environment. Neither the CPN model, nor the use case diagram in Figure 1, explicitly describe details of the software we are designing for the elevator controller. The elements of both the use case diagram and the CPN model are to be thought of as real-world elements, like a real motor, a real button, and a real door.

To design the elevator controller, we need to make two steps that will move us from the environment-level descriptions we have, to a description of the software. This will involve (1) derivation of a class diagram and (2) specification of behaviour, both of the individual class instances (intra-object behaviour in the sense of [4]) and of the communication between class instances (inter-object behaviour). We will address issue (1) in Section 4 and issue (2) in Section 5. For a further discussion of the importance of distinguishing between models of the environment and models of software, please refer to [8].

4 Class Diagram

A class diagram for the elevator controller is shown in Figure 3

The class diagram should be compared with the use case diagram of Figure 1. Each class in the class diagram, which has the same name as an actor in the use case diagram, is the software representation of that actor inside the elevator controller, e.g., instances of the `FloorButton` class represents the real-world, physical floor buttons. In addition to the classes representing actors from the use case diagram, the class diagram contains the classes `Cage`, `FloorButtonAllocation`, `CageButtonAllocation`, and `Initiator`.

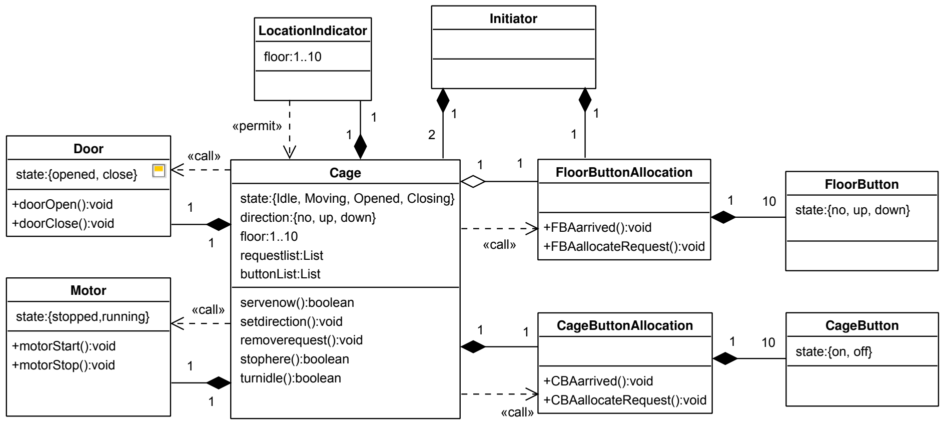


Figure 3: Class diagram for the elevator controller.

The elevator controller will contain two instances of the `Cage` class; each one represents one of the two real-world cages. The state of each cage object reflects the information about the real world that the controller needs in order to do its job. That information includes, e.g., knowledge of the position and direction of movement of the modelled cage. In general, attributes and operations of the classes in the class diagram are derived from the requirements-level CPN model.

The classes `FloorButtonAllocation` and `CageButtonAllocation` are used to model allocation of requests to cages. More specifically, they group the several cage and floor buttons and delegate the request handling to the respective button classes: `FloorButton` and `CageButton`. Taken together, they play a similar role as the `Handle Requests` module in Figure 2. There is exactly one instance of the `Initiator` class; its only aim is to create other objects inside the elevator controller. The `LocationIndicator` class plays a similar role to the `Move Up Down` module in Figure 2; its objects have direct access to the respective `Cage` object's private attributes and update the location indicators while the cage is in state `Moving`;

Finally, the `Motor` and `Door` objects receive commands from the controller; these commands are modelled by class operations.

5 Object-oriented CPN Model

From the requirements-level CPN model of Figure 2, we now move towards a design-level CPN model. An essential difference between these two models is that the requirements-level CPN model describes the desired behaviour of real-world entities whereas the design-level CPN model describes the desired behaviour of the software that must control the real-world entities. Thus, the two models are related, but different. The design-level CPN model describes the behaviour of each class of the class diagram, and the combined behaviour of the entire controller. This is made possible through the use of synchronous communication – modelled by transition fusion – and a set of object-oriented modelling idioms.

Figure 4 shows the CPN model for the `Motor` class (see Figure 3), which exemplifies the modelling of a class behaviour by a CPN. When defining the CPN for a class we use the tokens to carry the object reference plus all the object attributes, if any. As each object is modelled by a token, the CPN class model models the behaviour of all class instances. This means the CPN model is the object system model.

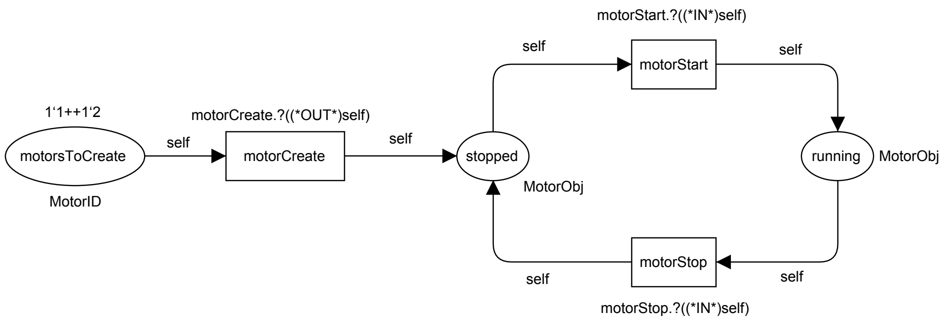


Figure 4: CPN for the `Motor` class, using channels.

Class `Motor` models the behaviour for a maximum of two objects, with two possible states: `stopped` and `running`. The objects are created by transition `motorCreate`. This transition has an associated receive channel, specified by the syntax *channelName.?(parameters)*, where the *?* stands for the receive part of the channel. In object-oriented terms, one can think of it as a *class operation* as it does not receive an object reference and, as in this case, it typically returns a new object reference (the `self` attribute). In the presented example, the `motorCreate` operation is called from transition `cageCreateBegin`, which is part of the CPN model for class `Cage` (top-left corner of Figure 6), which we will describe later.

The place `motorsToCreate` specifies that only two different objects can be created. For each firing of transition `motorCreate`, one `Motor` object is created. In this simple case, it consists of the variable `self`, which is the object self reference. When a `Motor` object is created its state is `stopped`. This is specified by place `stopped` in Figure 4. Later, after transition `motorStart` firing, a `Motor` object state can take the value `running`.

Each `Motor` object has two public operations, whose single effect is to change the object state: `motorStart` and `motorStop`. Each of these operations is modelled by a transition and an associated receive channel with the same name. An instance operation has the object reference as one of its parameters. This has the advantage of being similar to the syntax and semantics commonly found in object-oriented programming languages.

An operation call is specified by the send part of a channel and has the syntax *channelName.!(parameters)*. The channel's semantics is defined by the fusion of the two transitions (with the send and receive parts). The resulting transition has a guard which is the conjunction of both guards. As an example, Figure 5 shows a transition (named `Stop/motorStop`) with the equivalent semantics to the synchronous channel `motorStop` between transition `Stop` in the bottom-right corner of Figure 6 and transition `motorStop` in Figure 4.

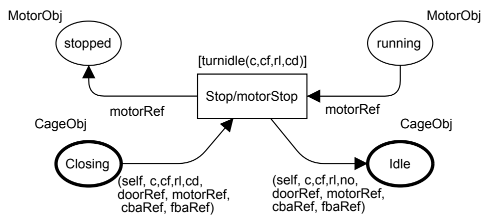


Figure 5: Transition fusion for channel `motorStop`.

Figure 6 shows the CPN model for class `Cage`. The requirements-level model in Figure 2 defined a `CAGE` data type with the form `(cageid, floor, requestlist, direction)`. For the class model, all these four elements are defined as attributes for `Cage` objects. We also define, as `Cage` objects' attributes, the references to the composite objects `CageButtonAllocation`, `LocationIndicator`, `Door`, and `Motor`, and the references to the aggregate object `FloorButtonAllocation`. Hence, each `Cage` object has the form `(self, cageid, floor, requestlist, direction, doorRef, motorRef, cbaRef, fbaRef)`. Besides the variable `self` for the objects' self reference, we use variables with the syntax *nameRef* for specifying references to other objects. The remaining variables are the ones used for the requirements level model in Figure 2.

The `Cage` class has the same basic structure as the `Do Cage Cycle` module in Figure 2. The differences are due not only to a more detailed specification, which is closer to the actual controller implementation, but also due to an increased modularity made possible by the use of synchronous channels.

Whereas the initial `Do Cage Cycle` module (Figure 2) assumes that both cages initially exist in the `Idle` state, the `Cage` class net models the cage objects creation explicitly. The operation `cageCreateBegin` creates a `Cage` object (and so it returns the new object reference), and also calls the create operations for all the composite objects. Differently, the reference for the aggregate `FloorButtonAllocation` object (`fbaRef`) is received from the object `Initiator` that creates the cage object.

The `Motor` class (see Figure 4) returns the object reference (`self`) through the channel `motorCreate`. This reference is then used by the `Cage`

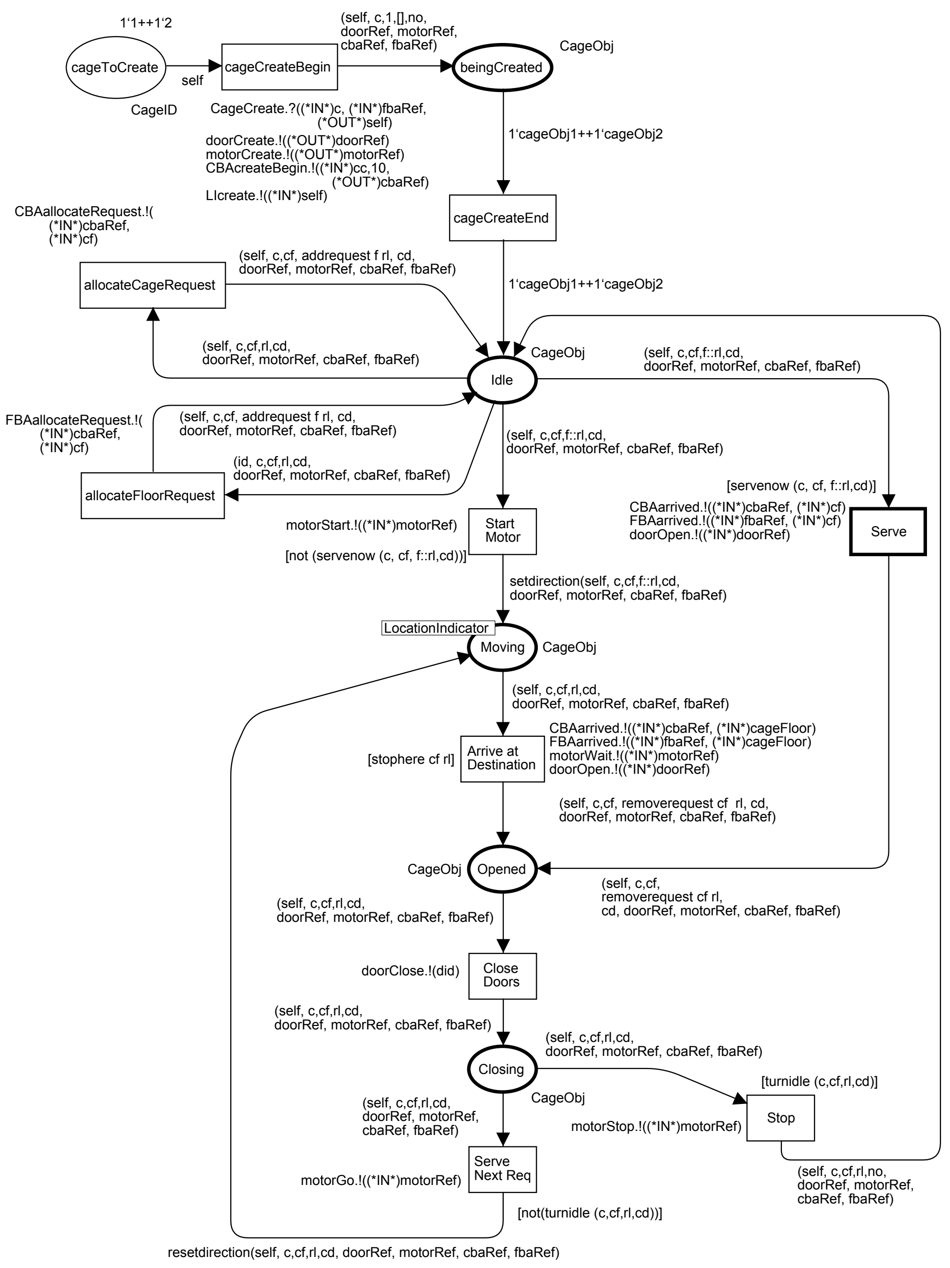

1'1++1'2
cageToCreate
self
CageID
cageCreateBegin
(self, c,1,[],no, doorRef, motorRef, cbaRef, fbaRef)
beingCreated
CageObj
CageCreate.?((*IN*)c, (*IN*)fbaRef, (*OUT*)self)
doorCreate.!((*OUT*)doorRef)
motorCreate.!((*OUT*)motorRef)
CBAcreateBegin.!((*IN*)cc,10, (*OUT*)cbaRef)
LIcreate.!((*IN*)self)
1'cageObj1++1'cageObj2
cageCreateEnd
1'cageObj1++1'cageObj2
CBAallocateRequest.!((*IN*)cbaRef, (*IN*)cf)
allocateCageRequest
(self, c,cf, addrequest f rl, cd, doorRef, motorRef, cbaRef, fbaRef)
(self, c,cf,rl,cd, doorRef, motorRef, cbaRef, fbaRef)
Idle
CageObj
(self, c,cf,f::rl,cd, doorRef, motorRef, cbaRef, fbaRef)
FBAallocateRequest.!((*IN*)cbaRef, (*IN*)cf)
(self, c,cf, addrequest f rl, cd, doorRef, motorRef, cbaRef, fbaRef)
(id, c,cf,rl,cd, doorRef, motorRef, cbaRef, fbaRef)
allocateFloorRequest
(self, c,cf,f::rl,cd, doorRef, motorRef, cbaRef, fbaRef)
[servenow (c, cf, f::rl,cd)]
CBAarrived.!((*IN*)cbaRef, (*IN*)cf)
FBAarrived.!((*IN*)fbaRef, (*IN*)cf)
doorOpen.!((*IN*)doorRef)
Serve
motorStart.!((*IN*)motorRef)
Start Motor
[not (servenow (c, cf, f::rl,cd))]
setdirection(self, c,cf,f::rl,cd, doorRef, motorRef, cbaRef, fbaRef)
LocationIndicator
Moving
CageObj
(self, c,cf,rl,cd, doorRef, motorRef, cbaRef, fbaRef)
CBAarrived.!((*IN*)cbaRef, (*IN*)cageFloor)
FBAarrived.!((*IN*)fbaRef, (*IN*)cageFloor)
motorWait.!((*IN*)motorRef)
doorOpen.!((*IN*)doorRef)
[stophere cf rl]
Arrive at Destination
(self, c,cf, removerequest cf rl, cd, doorRef, motorRef, cbaRef, fbaRef)
CageObj
Opened
(self, c,cf, removerequest cf rl, cd, doorRef, motorRef, cbaRef, fbaRef)
(self, c,cf,rl,cd, doorRef, motorRef, cbaRef, fbaRef)
doorClose.!(did)
Close Doors
(self, c,cf,rl,cd, doorRef, motorRef, cbaRef, fbaRef)
(self, c,cf,rl,cd, doorRef, motorRef, cbaRef, fbaRef)
Closing
[turnidle (c,cf,rl,cd)]
CageObj
(self, c,cf,rl,cd, doorRef, motorRef, cbaRef, fbaRef)
motorStop.!((*IN*)motorRef)
Stop
(self, c,cf,rl,no, doorRef, motorRef, cbaRef, fbaRef)
motorGo.!((*IN*)motorRef)
Serve Next Req
[not(turnidle (c,cf,rl,cd))]
resetdirection(self, c,cf,rl,cd, doorRef, motorRef, cbaRef, fbaRef)

Figure 6: CPN for the `Cage` class, using channels.

objects (it must evaluate to the same value as the `motorRef` variable in class `Cage`). The `Door` class is handled in the same way.

When compared to the `Do Cage Cycle` module, the `Cage` class has three main differences:

1. The `Move UpDown` module is replaced by the `LocationIndicator`

class.

2. The cage buttons requests are now handled by the `CageButtonAllocation` class together with the `CageButton` class; likewise, the floor buttons' requests are now handled by class `FloorButtonAllocation` together with `FloorButton`; channels associated to the `Arrive at Destination` and `Serve` transitions are now responsible for updating the buttons' state; additional transitions `allocateCageRequest` and `allocateFloorRequest` get the pending requests, through the associated channels, from the `CageButtonAllocation` class and the `FloorButtonAllocation` class, respectively.

3. The cage door and cage motor are now created as composite objects (in transition `cageCreate`); after, the `Motor` class objects change state through channels `motorStart` and `motorStop`; the `Door` objects change state through channels `doorOpen` and `doorClose`.

Compared with the requirements-level CPN model, the object-oriented design-level CPN model that we have just presented is a step towards code in an object-oriented programming language; we discuss this in more detail in the next section.

6 Towards an Implementation

The straightforward approach to execute a CPN model is to use a simulator like CPN Tools or Renew [20]. Yet, this not always possible as the model may have to be run on a platform where the simulator is not available or has insufficient performance. The alternative is a code generation approach as this allows code optimisations adaptable to each software and hardware platform.

One way to implement the CPN model is to translate it to a state machine. Yet, this state machine will probably be too large to be useful in practice: in fact, through that approach we will be implementing the complete state space. In many cases, this is not a solution, as the state space is simply too large. Yet, for small enough state spaces this can be an efficient solution, especially for hardware platforms, which have limited memory resources but are able to efficiently execute state machines. These state machines can be coded in ANSI C, which is usually available for embedded operating systems. This path allows the design, in a high-level specification language, of object-oriented models, which are finally implemented as a state-machine. In this sense, the CPN model can be seen as a higher-level alternative to a state-machine (or statechart) based model.

The alternative approach is to generate an interpreter for the model. This could be implemented as two distinct packages: (1) the net structure specification; (2) the net executor.

The following guidelines allow a straightforward implementation for the net structure, in a general object-oriented programming language (e.g. Java or C++):

- Each Colour Type is defined as a class.
- Each CPN page is implemented as a class.
- Each place, transition, and arc, is an object of class Place, Transition, and Arc, respectively.
- Each CPN class contains as attributes the respective page variables, transitions and places.
- Each place object contains a bag of elements of the data type associated to the place in the net model (the place colour).
- Each transition object has four methods: (1) one to test the transition state (*enabled* or *not enabled*); (2) one to fire the transition, accordingly changing the markings of the respective input and output places; (3) one implementing the guard; (4) one implementing the code segment.
- Each arc object has a reference to a place object and a reference to a transition object; it also contains a method implementing the arc expression and returning a value of the place colour.
- Each port place, in a hierarchical CPN, is a reference attribute to a place object in another class.
- Each place fusion, again for a hierarchical CPN, is coded as a place object and a set of place object references.

For each execution step, the net executor package executes three substeps: (1) calculates the set of enabled transitions; (2) chooses a subset to fire; (3) fires the transitions in the chosen subset.

For large nets, the first sub-step can impose a significant computation delay. The binding computation has the potential to become the execution bottleneck, although it can be minimised by testing only the transitions where, at least, one input place has changed its marking in the previous step.

The usual memory versus speed compromise clearly shows up when implementing CPNs: either the CPN compactness compared to a low-level Petri net, has to go away through the total unfolding of the model, in the form of a state-machine; or we implement a direct execution implying the binding's computation, which is slower although more memory efficient.

7 Related Work

The channels for synchronous communication were first proposed by Christensen and Hansen [2]. They bring the transition fusion concept to CPNs. In particular, transitions are able to communicate complex values. By allowing synchronous communication modelling in CPNs plus data communication, synchronous channels offer an effective way to create more compact and readable models. The initial proposal by Christensen and Hansen is totally symmetric, namely there is no sender and receiver sides for each channel.

Kummer proposed a direction of invocation to each channel [15]: we get *sender* and *receiver* sides for each channel. Yet, the parameters are still bidirectional. For example, one net (using a send channel in one of its transitions) can pass a value through one parameter and receive a value from another parameter. In [1] it was proposed to further disambiguate this bidirectional nature for channel parameters by qualifying the parameters as `IN`, `OUT`, or `INOUT`. These give channels the usual parameter passing semantics. Together with the direction of invocation, channels become closer to method invocation. Here, we used these qualifiers inside Standard ML comments, as they are not supported by CPN Tools. Presently, the same happens to synchronous channels.

It is clear that if we want to generate executable, autonomous code from a CPN model, we should use the inscription language as the target programming language. CPN Tools uses Standard ML as the inscription language. Hence, it is easier to generate Standard ML code for model execution [17, 13]. Another approach is to use a code-level library, in the target language, supporting the net execution and add it to a net structure specification also in the target language (e.g. [18]). A general object-oriented language can also be used. A significant example is the Renew tool [20], which supports an extension to CPNs, is written in Java, and uses Java as the inscription language.

8 Conclusions and Future Work

CPN models as we have presented them in this paper might candidate to be used in object-oriented analysis and design, e.g., in development projects tailored from the widely used *Rational Unified Process (RUP)* [14]. RUP emphasises the use of UML models as key artifacts in software development. If a deviation from UML is acceptable by the project stakeholders, creating and executing CPN models fit well in the elaboration phase or the inception phase of RUP. However, it will take a major effort, of course, to transfer the specific observations done in the small case study of this paper into generally applicable guidelines for the interplay between traditional RUP artifacts like use case diagrams and class diagrams and various kinds of

CPN models.

Compared to many other graphical languages for behaviour description, CPNs have the following advantages: (1) they have a precise syntax and semantics; (2) they are executable; (3) they are verifiable; (4) they scale well as testified by numerous examples in literature.

CPN's main drawbacks seem to be (1) the lack of clear connections to other system views, namely to a structure diagram; (2) their efficient implementation. The first point was addressed here through the use of channels for synchronous communication. In particular, transitions are able to communicate complex values. By allowing synchronous communication modelling in CPNs plus data communication, synchronous channels improve models modularity and readability. In particular, when used together with the presented idioms, they allow the designer to think and model in object-oriented terms (see also [1]). In this way, they provide a one-to-one relationship between classes and the modules of a CPN model, thus establishing a clear connection to class diagrams.

The mapping between object-oriented CPN models and executable autonomous code is still a subject for further work. In particular, the object-oriented CPN model must be able to specify the platform specific code and data. Code segments associated to transition firing and CPNs data declarations may facilitate this. Yet, as the literature and applications on CPNs model implementation is scarce, it is still debatable if this is sufficient. We attribute this situation to the Petri net's community traditional bias towards model verification and analysis. This comes probably from the deeply formal and abstract Petri nets' origin, as a mathematical object. Yet, CPNs, even without further semantic extensions, are a class of Petri nets ready to be applied to object-oriented design.

References

[1] J. P. Barros and L. Gomes. On the Use of Coloured Petri nets for Object-Oriented Design. In Jordi Cortadella and Wolfgang Reisig, editors, *ICATPN 2004, Bologna, Italy, June 21-25, 2004*, volume 3099 of *LNCS*, pages 117–136. Springer, jun 2004.

[2] S. Christensen and N. D. Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. In R. Valette, editor, *ICATPN 1994, Zaragoza, Spain*, volume 815 of *LNCS*, pages 159–178. Springer, jun 1994.

[3] CPN Tools. CPN Tools homepage. `http://wiki.daimi.au.dk/cpntools`, 2004.

[4] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001.

[5] J. Desel and G. Juhás. *What is a Petri net*, volume 2128 of *LNCS*, pages 1–25. Springer, 2001.

[6] M. Elkoutbi and R. K. Keller. User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets. In *Proceedings of 21st Petri Nets Conference*, volume 1825 of *LNCS*, pages 166–186, Aarhus, Denmark, 2000. Springer.

[7] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[8] M. Jackson. *Problem Frames — Analyzing and Structuring Software Development Problems.* Addison-Wesley, 2001.

[9] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 1-3.* Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1992-97.

[10] J. B. Jørgensen. CPN Models as Enhancements to a Traditional Software Specification for an Elevator Controller. In *Proc. of 3rd Workshop on Modelling of Objects, Components, and Agents (MOCA'04)*, pages 99–116, Aarhus, Denmark, 2004. University of Aarhus. Technical report.

[11] J. B. Jørgensen and C. Bossen. Executable Use Cases: Requirements for a Pervasive Health Care System. *IEEE Software*, 21(2):34–41, 2004.

[12] J. B. Jørgensen and S. Christensen. Executable Design Models for a Pervasive Healthcare Middleware System. In *Proc. of 5th UML Conference*, volume 2460 of *LNCS*, pages 140–149, Dresden, Germany, 2002. Springer.

[13] L. M. Kristensen and S. Christensen. Implementing Coloured Petri Nets Using a Functional Programming Language. *Higher-Order and Symbolic Computation*, 17(3):207–243, 2004.

[14] P. Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley, 1999.

[15] O. Kummer. Simulating Synchronous Channels and Net Instances. In Jörg Desel, Peter Kemper, Ekkart Kindler, and Andreas Oberweis, editors, *Forschungsbericht Nr. 694: 5. Workshop Algorithmen und Werkzeuge für Petrinetze*, Forschungsbericht Nr. 694, pages 73–78. Fachbereich Informatik, Universität Dortmund, 1998.

[16] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML.* MIT Press, 1997.

[17] K. H. Mortensen. Automatic Code Generation from Coloured Petri Nets for an Access Control System. In *Kurt Jensen (ed.): Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN, Aarhus, Denmark*, pages 41–58, October 1999.

[18] C. Reinke. Haskell-Coloured Petri Nets. In *IFL '99: Selected Papers from the 11th International Workshop on Implementation of Functional Languages*, pages 165–180. Springer-Verlag, 2000.

[19] W. Reisig. *Petri nets: an Introduction.* Springer-Verlag New York, Inc., 1985.

[20] RENEW. The Reference Net Workshop homepage. `http://www.renew.de/`, 2004.

[21] R. J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML.* Morgan Kaufmann, 2003.

Transforming Coloured Petri Nets to Counter Systems for Parametric Verification: A Stop-and-Wait Protocol Case Study [*]

Jonathan Billington and Guy Edward Gallasch
Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia
Mawson Lakes, SA, 5095, AUSTRALIA
Jonathan.Billington@unisa.edu.au
Guy.Gallasch@postgrads.unisa.edu.au

Laure Petrucci
LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, FRANCE
petrucci@lipn.univ-paris13.fr

Abstract

Protocols may contain parameters that are chosen from a wide range. In some cases we would like our analysis results to apply to an arbitrary upper limit on a parameter value, such as the maximum number of retransmissions. In this case we have an infinite family of finite state systems. This makes their verification difficult. However, techniques and tools are being developed for the verification of parametric and infinite state systems. We explore the use of one such tool, FAST, for verifying several properties of the stop-and-wait class of protocols, where the maximum number of retransmissions and the maximum sequence number are consider parameters. We are also interested in using expressive languages for representing protocols such as Coloured Petri nets (CPNs). FAST's foundation is counter systems, which are automata whose states are a vector of non-negative integers, with operations limited to Presburger arithmetic. We therefore also present some first steps in transforming CPNs to counter systems in the context of stop-and-wait protocols operating over unbounded FIFO channels.

[*]This work is supported by the Australian Research Council Linkage International grant LX04544639, and the French-Australian Science and Technology programme FR040062.

1 Introduction

1.1 Background and Motivation

The design and development of computer communication protocols is central to the development of embedded and pervasive computing systems which nearly always involve the co-operation of distributed components. It is important that protocols behave according to their requirements, since their failure can have serious consequences particularly for life critical or financially sensitive applications. Being able to verify that protocols behave correctly is a significant challenge since they usually include a number of parameters (such as a maximum sequence number, flow and congestion control window sizes, and the maximum number of retransmissions) that may be chosen to suit the operating environment, and may vary widely. Thus we would like to consider a class of protocols where the parameters can take any value within their range, and verify their correctness for all values of the parameters. Sometimes the ranges for these parameters are unbounded, giving rise to an infinite family of state spaces, one for each value of the parameter. At other times no limit may be placed on the value of a parameter (e.g. the number of times a packet can be retransmitted) which may result in an infinite state space.

The approach we use to tackle this problem is that of model driven development. The first step is to develop a formal model of our system which we then analyse either using tools or if they fail then by hand or a combination of both. The model is normally at the design level and the proofs are intended to show that the design satisfies the requirements of the system. This is rather important because removing errors at the design stage is very cost effective in the development of systems compared with removing errors in the implementation using testing. The effect is even more pronounced if the errors are discovered after the product has been released. The development of the model and its analysis can also increase the level of understanding of the requirements. Further, if the model is executable it can be used in fast prototyping of system specifications. This also increases the designer's and customer's understanding of requirements, which is widely acknowledged as a problematic area in software development.

In previous work [8] we summarised a protocol verification methodology based on Coloured Petri nets [18] and finite state automata. (Coloured Petri Nets (CPNs) are an executable modelling language with a formal semantics, based on Petri nets and the ML functional programming language.) This methodology uses state space methods and has been applied successfully for finite state systems, for small values of parameters. Techniques (such as partial orders, BDDs, and the sweep-line method) for alleviating the state space explosion problem [28] help to extend the method to larger ranges of parameters, but cannot handle large or unbounded values.

In [8], the methodology is illustrated using a stop-and-wait Protocol (SWP) [25, 22] which involves two parameters: the maximum sequence number, MaxSeqNb; and the maximum number of retransmissions, MaxRetrans. From a modelling point of view, the values of these parameters may be chosen arbitrarily. We would thus like to prove that the SWP class is correct for any values of $\mathsf{MaxSeqNb} \geq 1$ and $\mathsf{MaxRetrans} \geq 0$. This becomes impossible using finite state techniques, as we need to consider an infinite number of increasingly larger finite state spaces. For FIFO channels (either lossy or lossless), a hand proof is given in [8] that shows that the number of messages in the message channel (and the number of acks in the acknowledgement channel) has a least upper bound of 2MaxRetrans + 1, for any positive value of MaxSeqNb, and any non-negative value of MaxRetrans. For other properties, such as verifying that the protocol conforms to its service of alternating send and receive events, the standard methodology was used for a range of parameter values ($0 < \mathsf{MaxSeqNb} < 1024$, $0 \leq \mathsf{MaxRetrans} \leq 4$), but no general result was obtained. This has motivated us to search for methods that will handle unbounded parameters and provide some degree of automation.

This paper addresses the unbounded parameter problem by using a tool called FAST (Fast Acceleration of Symbolic Transition systems) [3], based on counter systems [17]. FAST performs symbolic analysis of infinite state systems by using *accelerations* (*meta-transitions*) to encode an arbitrary number of iterations of sequences of actions within the system. Parameters can be input as variables that are not constrained, and hence automated parametric verification of systems may be possible. However, we face two difficulties using this tool. Firstly, FAST's input language is based on counter systems (CS), whereas we would like to use the much more expressive language of CPNs. CS can model Place/Transition nets augmented with special arc types such as inhibitors [15], but as far as we are aware no attempt has previously been made to translate CPNs to CS. Secondly, FAST provides a semi-algorithm, which is not guaranteed to terminate. Hence we can never be sure the verification will succeed.

The purpose of our work is thus to explore the potential of FAST for the parametric verification of communication protocols which have been previously modelled using CPNs. This paper investigates the class of stop-and-wait protocols. This is because they require parametric verification and are the simplest representative example of the class of protocols which provide flow control and bit error recovery that are used in practice, such as in the data link and transport layers of communication protocol architectures. We slightly revise our CPN model in [8] to make it easier to translate to a counter system. We find that translating CPN places representing states, stored sequence numbers and the retransmission counter is straightforward, but queues are more of a challenge. We are able to use 4 integer variables to represent the FIFO queue, due to the operation of the SWP. The conditions that are required for the queue model to be valid are checked using FAST, as well as the following properties: channel bounds; deadlocks; the stop-and-wait property; in-sequence delivery; and message loss and duplication; for both lossless and lossy FIFO channels.

1.2 Related Work

The simplest SWP restricts its sequence numbers to 0 and 1 and is known as the Alternating Bit protocol (ABP) [5]. The ABP and its extensions (e.g. [12]) have been used extensively in the literature as case studies (e.g. [21]). Often such papers demonstrate in various ways whether the ABP works as expected over (lossy or lossless) FIFO channels [24, 9], investigate performance [23, 20], demonstrate new tools [9], or illustrate verification methodologies [14], the application of formal description techniques [27], new modelling languages or derivatives of existing languages [24, 23, 20]. However, none of these papers address the issue of parametric verification of the ABP (i.e. for arbitrary values of MaxRetrans.)

More recently there has been work in the area of symbolic verification of the ABP. Valmari and his co-workers (e.g. [29]) promote a behavioural fixed point method and compositional techniques for the verification of parametric systems. In [29] a variant of the ABP using limited retransmission, i.e. where there is an arbitrary bound (e.g. MaxRetrans) on the number of retransmissions, is verified using Valmari's CFFD equivalence. There are several differences with our work. Perhaps the most significant is that the channels are limited to holding only one message or acknowledgement at a time, whereas ours are unbounded FIFO queues. Valmari [29] concedes this to be a much more difficult problem. Valmari's method relies on defining a separate counter process which needs to be synchronised (using parallel composition) with the sender logic, which has 18 states. The counter itself is a recursive parallel composition of counter cells. The receiver is a relatively straightforward 6 state process. The ack channel is given as a 3 state process, but the data channel is more complex and not given explicitly in the paper. To obtain the model, all these processes need to be synchronised with parallel composition. In contrast our CPN model integrates all these aspects in the one model, and extends the model to include unbounded FIFO queues and sequence numbers with an arbitrary maximum sequence number as a parameter. However, our model does not have explicit communication with the users (but relies on the send and (non-duplicate) receive transitions to be considered as synchronised communication with the user) and does not consider reporting errors to the user. We see no problem in extending our model to include these features, however, our aim is to illustrate the use of FAST in analysing parameterised CPN models, rather than a direct comparison with a particular ABP variant.

The ABP and another variant called the Bounded Retransmission Protocol are used in [2] to demonstrate a symbolic verification methodology [1]. TReX (Tool for Reachability Analysis of Complex Systems) [26] was used to implement this methodology in [2]. The content of unbounded lossy FIFO channels is modelled by (a restricted class of) regular expressions thus providing a symbolic representation of the channels. Similar to FAST, an acceleration technique is used. This allows a small symbolic state space to be calculated based on the states of the sender and receiver ABP processes. They verify that the ABP conforms to its service of alternating sends and receives, using the Aldebaran tool [11] for finite state automata. The maximum number of retransmissions was considered to be unlimited giving rise to a single, infinite-state model. This differs from our approach of modelling MaxRetrans as a parameter and thus having an infinite number of finite-state models, one for each parameter value. As mentioned above, we also model an arbitrary maximum sequence number, rather than being limited to a maximum sequence number of 1.

1.3 Contribution and Organisation

This paper provides two contributions. Firstly, we believe it is the first time that parametric verification of the stop-and-wait protocol class operating over unbounded FIFO channels has been undertaken where MaxRetrans has been modelled as a parameter. We are able to verify the SWP for arbitrary values of MaxRetrans for small values of MaxSeqNb (i.e. 1 to 5), for an extensive range of safety properties. Secondly, we provide some steps towards a method for translating CPNs into counter systems.

The rest of the paper is organised as follows. Section 2 describes the stop-and-wait protocol using a Coloured Petri net model. Counter Systems are introduced in Section 3 which also describes a methodology for translating a CPN model into a CS. This methodology is applied in Section 4 to the SWP CPN of Section 2. The expected properties of the SWP are described in Section 5. After introducing FAST, we analyse the SWP CS in Section 6. Section 7 provides concluding remarks and identifies areas of future work.

2 Stop-and-Wait Class of Protocols: A CPN Model

We explain the class of stop-and-wait protocols by providing a parameterised Coloured Petri Net (CPN) model of it as shown in Figs. 1 and 2, which were created using Design/CPN [13]. Essentially three changes are made to the CPN model presented in [7, 8]:

- in the sender, one place (instead of two) is used to store its states, so that the colour set Sender comprises two states: s_ready and wait_ack;
- one place (receiver_state) is used in the receiver to store its states;
- a new place in the receiver stores its current sequence number; and
- arc inscriptions are revised accordingly.

This makes the CPN diagram more compact and provides a consistent modelling style. Control flow is indicated by bold arcs.

The protocol operates between a sender, shown on the left of Fig. 1 and a receiver shown on the right. The communication medium (Network) is represented by two lossy FIFO queues, one for each direction of message flow. The queues are modelled by using a list type for places mess_channel and ack_channel, adding

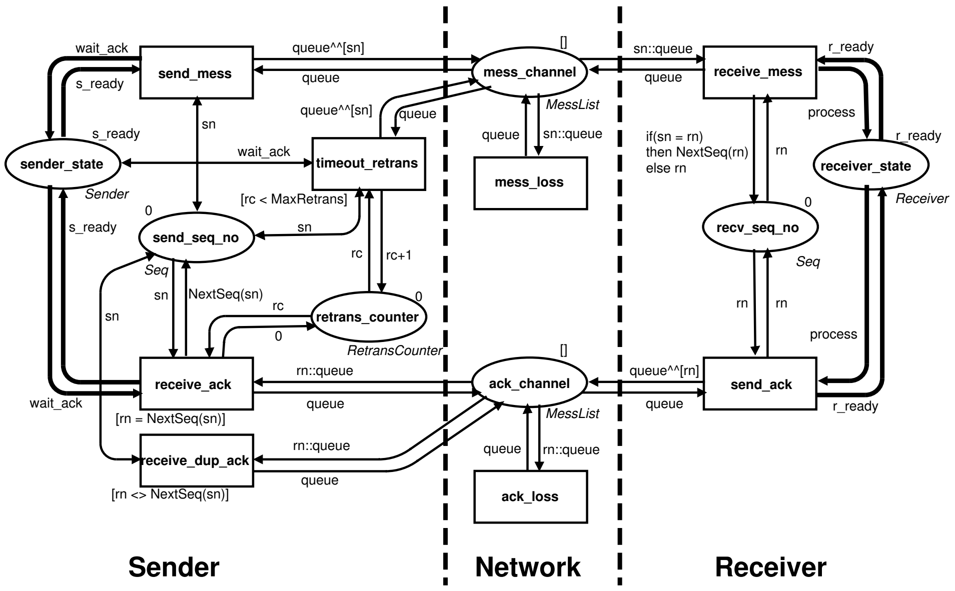


Figure 1: CPN of the SWP operating over a lossy FIFO channel.

```
val MaxRetrans = 1;
val MaxSeqNb = 1;

color Sender = with s_ready | wait_ack;
color Receiver = with r_ready | process;
color Seq = int with 0..MaxSeqNb;
color RetransCounter = int with 0..MaxRetrans;
color Message = Seq;
color MessList = list Message;

var sn,rn   : Seq;
var rc      : RetransCounter;
var queue   : MessList;

fun NextSeq(n) = if(n = MaxSeqNb) then 0 else n+1;
```

Figure 2: Global Declarations for the Stop-and-Wait Protocol CPN.

messages to the end of the queue (using the operator ^^) and removing messages from the head of the queue (using ::). Loss is represented by arbitrarily removing the head of the queue.

The protocol is implemented by the sender and receiver procedures. The sender has two states: s_ready and wait_ack, with the current state stored in place sender_state. When ready, it sends a message (transition send_mess) and waits for an acknowledgement before sending the next message (hence, stop-and-wait). To overcome the possibility that the message has been lost, the sender sets a timer running on sending a message, and if it expires before receiving the acknowledgement (receive_ack), the message is retransmitted (transition timeout_retrans) and the timer is set running again. However, the acknowledgement could be lost even though the message had been received. In this case, the receiver needs to detect and discard duplicate messages. To do this, a sequence number is associated with each message, and stored by the sender (place send_seq_no). In the CPN model, messages are represented by their sequence number only, as data is not used in the procedures. The receiver also stores a sequence number (place recv_seq_no) which is used to detect duplicates. The sequence number space is finite, but allows for any range of consecutive integers, starting from zero. In our model we use the parameter MaxSeqNb to repesent the maximum value of the sequence number space. If a new message arrives at the receiver (transition receive_mess with sn=rn), the sequence number is incremented, modulo MaxSeqNb (NextSeq(rn)), whereas if a duplicate is detected (sn≠rn) the sequence number remains the same. The sequence number in the receiver represents the next message to be received, and this is the value that is sent back to the sender, as an acknowledgement. Acknowledgements are returned on receipt of each message, whether or not it is a duplicate. (This is required to recover from loss of acknowledgements.) To model flow control, we have two states in the receiver (r_ready and process), which allow the sending of the acknowledgement to be asynchronous with the receipt of a message. The current state is stored in place receiver_state.

While waiting for an acknowledgement, the sender may continue to retransmit messages, until it reaches a preset limit (MaxRetrans). It then gives up hope of the message getting through and passes control to a management entity for higher level recovery (not modelled). If an acknowledgement arrives before this, indicating that the message has been received (rn=NextSeq(sn)) transition receive_ack increments the send sequence number and returns the sender to ready, allowing the next message to be sent. Duplicate acknowledgements are discarded by the sender (receive_dup_ack) at any time.

3 Mapping the CPN Model to a Counter System

As we aim at obtaining an extensive set of analysis results on the stop-and-wait protocol, parametric analysis is desirable. It can be achieved using tools such as FAST [3]. FAST operates on *counter systems*, so it is necessary to transform our CPN model into a CS. This is straightforward for Petri nets, even with extended arcs [6, 4] but requires enhancement for CPNs.

Counter systems are automata extended with *unbounded integer variables*. FAST uses *accelerations* (sometimes called *meta-transitions*) to enable it to calculate the exact effect of iterating a behavioural loop an arbitrary number of times, and produces a symbolic occurrence graph representing the infinite state system. Details on counter systems and the theory behind FAST can be found in [17, 10, 30, 19].

The places of a CPN are transformed into a set of counter system variables and a single counter system state. This transformation is straightforward if the types of the places are or can be mapped to integers (e.g. enumerated types). If a place p has a type $\mathit{Type}(p)$ that can be mapped to the integers by an injective mapping, $I_p : \mathit{Type}(p) \to \mathbb{N}$, and p always contains one token ($\forall M \in [M_0\rangle, |M(p)| = 1$), then we can create an integer variable v_p in the CS, that takes the values of the token in the place transformed by I_p for each marking. This is the case both for the places in the sender and those in the receiver of our SWP CPN model.

However, the stop-and-wait protocol uses two FIFO queues: one for messages and one for acknowledgements, represented by places mess_channel and ack_channel both typed by a message list, where messages are represented by sequence numbers (integers). These queues can be any size, depending on the maximum number of retransmissions [8]. The values of the sequence numbers depend on the MaxSeqNb parameter. Because the sequence numbers are integers we can store the value of a queue item in a variable, and the number of queue items of that value in an associated variable. As long as sequence numbers do not wrap, we can always remove the item with the 'smallest' value from the queue and hence maintain FIFO order. However, this will require an unbounded number of variables in the general case, but not if the queue can only contain a finite number of values at any one time. For example, if the queue can contain only one message value at a time, then it can be represented by two variables: one storing the message value and a second storing the number of messages in the queue.

For the SWP operating over FIFO channels it turns out that there can be at most two different messages (represented by their sequence numbers) in the queue at any one time *and* that the messages of the same type are contiguous in the queue. Thus the queue is of the form mess1*mess2*. (Before doing the analysis, this property is a conjecture, so we must check that this property holds as a first step in validating the model.) Therefore, the queue can be modelled using four variables:

- Old is the smallest/oldest sequence number (modulo MaxSeqNb) that is in the queue;
- New is the latest sequence number that was put in the queue;
- NbOld is the number of messages with sequence number Old;
- NbNew is the number of messages with sequence number New.

Now, we will explain how to add messages to and remove messages from the queue. We will also show that this is done in a consistent manner.

The queue can contain:

1. *no message*. Hence NbOld $=$ NbNew $= 0$;
2. *one type of message*. Then, Old $=$ New and NbOld $=$ NbNew $\neq 0$;
3. *two types of message*. Thus, Old $\neq$ New, NbOld $\neq 0$ and NbNew $\neq 0$.

In the following, a prime denotes the value of the variable after an action has been performed. Variables that do not change are not mentioned.

First, consider adding a message with sequence number mess. If the queue is in state (numbered as above):

1. The new message is the only one. Therefore, after adding the message: $\mathsf{Old'} = \mathsf{New'} = \mathsf{mess}$ and $\mathsf{NbOld'} = \mathsf{NbNew'} = 1$. This is consistent with the above statement for a queue having a single message, hence containing only one type of message;
2. The new message can be either:
   - of the same type as those already in the queue. Then, after adding the new message, we have: $\mathsf{NbOld'} = \mathsf{NbNew'} = \mathsf{NbOld} + 1 (= \mathsf{NbNew} + 1)$. This is consistent with the queue containing a single type of message;
   - of a new type. Thus, $\mathsf{New'} = \mathsf{mess}$ and $\mathsf{NbNew'} = 1$. This is consistent with the queue now having two types of message.
3. In this case, only a New message (i.e. a duplicate) can be added to the queue and hence $\mathsf{NbNew'} = \mathsf{NbNew} + 1$. This is consistent with the queue containing exactly two types of message.

Now, we explain how to remove a message mess. If the queue is in state:

1. It is empty, so this case should never occur as there is nothing to consume;
2. The message consumed is of the single type in the queue. Hence: $\mathsf{mess} = \mathsf{Old} = \mathsf{New}$ and $\mathsf{NbOld'} = \mathsf{NbNew'} = \mathsf{NbOld} - 1 = \mathsf{NbNew} - 1$. Note that the resulting queue can either contain messages of the same single type or no message at all;
3. The message consumed can be of type either New or Old. Both cases can be handled in a similar manner, but in this paper, the queues considered are FIFO. Therefore, the message consumed is the oldest in the queue, i.e. $\mathsf{mess} = \mathsf{Old}$. Then two cases can be considered:
   - The message consumed is the last one of type Old, i.e. $\mathsf{NbOld} = 1$. Then the resulting queue contains a single type of message, $\mathsf{Old'} = \mathsf{New}$ and $\mathsf{NbOld'} = \mathsf{NbNew}$;
   - There are several messages of type Old in the queue. Then, $\mathsf{NbOld'} = \mathsf{NbOld} - 1$.

4 The SWP CS Model

The CPN model of the stop-and-wait protocol can now be transformed into a counter system by application of the techniques from the previous section.

4.1 SWP CS Variables

We first start with mapping the places of the SWP CPN to CS variables.

sender_state can take two values, i.e. s_ready or wait_ack. It is coded, in the CS, using a variable SState, with values 1 and 0 respectively;

send_seq_no becomes a variable SSeqNb, containing the last not-acknowledged sequence number;

retrans_counter is a variable Retrans, counting the number of retransmissions that have occurred for message number SSeqNb;

receiver_state can take two different values, i.e. r_ready or process. It is represented using a variable RState, with values 1 and 0, respectively;

recv_seq_no becomes a variable RSeqNb, containing the number of the next expected message;

mess_channel is modelled in the counter system using, as explained before, 4 variables MCOld, MCNew, NbMCOld and NbMCNew. They represent respectively, the sequence number of the message in the channel that was put first, the sequence number of the message in the channel that was put last, and the numbers of such messages;

ack_channel is modelled similarly with variables ACOld, NbACOld, ACNew and NbACNew.

Two other variables are needed for the parameters of the system: the maximum sequence number MaxSeqNb and the maximum number of retransmissions MaxRetrans.

4.2 SWP CS Transitions

When modelling the transitions, we must ensure that all possible cases are taken into account. In fact, we will only include firable transitions in the model, and not transitions that can never occur (dead transitions). It is important to reduce the number of transitions as in our case transition compositions (see Section 6.2) depend on the square of the number of transitions, thus increasing the computation time significantly. However we will check in Section 6 that this is the case, to ensure that nothing is missing. The transitions operate on the 4 variables describing queues. The wrapping from MaxSeqNb to 0 must be taken into account. The value of the other variables are changed as described in the CPN model.

4.3 The Stop-and-Wait protocol CS Model

Here, we show an excerpt of the SWP CS model, illustrating FAST model input. The model describes in a natural way the counter system to analyse. It comprises the integer variables identified above, the single state marking of the counter system obtained from the Petri net, and specifications of the transitions. Each transition is described by its source and destination states (from and to fields), which is here always the state marking. A guard is associated with each transition, giving an enabling condition on the values of the variables. The effect of the transition is given in action, describing how the values of variables are changed when the transition occurs. The symbol && indicates logical AND, || represents logical OR, ! indicates negation, and the prime notation is as defined in Section 3.

```
model SWP {
  var SState, SSeqNb, Retrans, MaxRetrans, MCOld, MCNew, NbMCOld, NbMCNew,
      ACOld, ACNew, NbACOld, NbACNew, RSeqNb, RState, MaxSeqNb;
  states marking;

// send: case new message with no message in queue
  transition sendM1 := {
    from   := marking;
    to     := marking;
    guard  := SState=1 && NbMCOld=0;
    action := SState'=0, MCNew'=SSeqNb, NbMCNew'=1, MCOld'=SSeqNb, NbMCOld'=1; };
// receive duplicate: case message with seq nb MCNew = MCOld
  transition receiveM1 := {
    from   := marking;
    to     := marking;
    guard  := RState=1 && NbMCOld>0 && !(MCOld=RSeqNb) && MCOld=MCNew;
    action := RState'=0, NbMCOld'= NbMCOld-1, NbMCNew'=NbMCNew-1; };
// receive duplicate ack: case ack with seq nb ACNew = ACOld
  transition recdupack1 := {
    from   := marking;
    to     := marking;
    guard  := NbACOld>0 && ACOld=ACNew && ((SSeqNb=MaxSeqNb && !(ACOld=0))
               || (SSeqNb<MaxSeqNb && !(ACOld=SSeqNb+1)));
    action := NbACOld'=NbACOld-1, NbACNew'=NbACNew-1; };
// receive expected ack
  transition recack := {
    from   := marking;
    to     := marking;
    guard  := NbACOld>0 && ACOld=ACNew && SState=0 && ((SSeqNb=MaxSeqNb &&
               ACOld=0) || (SSeqNb<MaxSeqNb && ACOld=SSeqNb+1));
    action := NbACOld'=NbACOld-1, NbACNew'=NbACNew-1, SState'=1,
              Retrans'=0, SSeqNb'=ACOld; };
...
}
```

Let us explain the declaration of transition sendM1. It starts and ends in the counter system state marking. The guard means that the sender sends a message only if it is ready to do so (SState=1) and the case handled by this transition is when the message channel is empty (NbMCOld=0). When these conditions are met, the transition can be fired and the action occurs, leading to a state were the sender is waiting for an acknowledgement (SState'=0), and the queue, containing only one message, hence both old and new, is updated (MCNew'=MCOld'=SSeqNb, NbMCNew'=NbMCOld'=1).

5 Required Properties of the CS Model

The model of the SWP should satisfy several properties, which are of two kinds: the properties ensuring that our translation from the CPN model to the counter system is sound, i.e. all the assumptions made are valid; and the properties that the protocol itself should satisfy.

5.1 Model Soundness

For the model to be sound, we need to verify the modelling assumptions. Our model is correct if both the message and acknowledgement channels: contain no more than two different types of message, where the 'type' of the message refers to its sequence number (i.e. Old and New from Section 3); and all messages of the same type are contiguous in the queue (i.e. the contents of the queue is of the form Old*New). To verify this, we check that if there are already two types of message in the queue (i.e. Old and New), only transitions which can add a New message are enabled.

We also verify the completeness of the model, i.e. that all the relevant cases are taken into account by transitions. Hence, all cases that are not explicitly described by the guards can never occur. This is done by verifying that there is no reachable marking that enables similar transitions, but with different conditions concerning the channel contents.

5.2 SWP Properties

We wish to prove the following SWP properties:

Consecutive sequence numbers If there are different types of message in a channel, they have consecutive numbers. Hence:

$$\mathsf{MCOld} \neq \mathsf{MCNew} \Rightarrow (\mathsf{MCNew} = \mathsf{MCOld} + 1 \vee (\mathsf{MCNew} = 0 \wedge \mathsf{MCOld} = \mathsf{MaxSeqNb}))$$
$$\mathsf{ACOld} \neq \mathsf{ACNew} \Rightarrow (\mathsf{ACNew} = \mathsf{ACOld} + 1 \vee (\mathsf{ACNew} = 0 \wedge \mathsf{ACOld} = \mathsf{MaxSeqNb}))$$

Number of messages in channels The lowest upper bounds for the number of messages in both channels, and the lowest upper bound on the total number of messages (i.e. messages plus acknowledgements) is $2\mathsf{MaxRetrans} + 1$. This is checked by counting the messages in the channels. The number of messages in the message channel is:

$$Nb\_Messages = \text{if } \mathsf{MCOld} \neq \mathsf{MCNew} \text{ then} \mathsf{NbMCOld} + \mathsf{NbMCNew} \text{ else } \mathsf{NbMCOld}$$

Hence, for the message channel:

$$Nb\_Messages \leq 2\mathsf{MaxRetrans} + 1$$

should hold over all reachable markings, but

$$Nb\_Messages \leq 2\mathsf{MaxRetrans}$$

should not. Similarly for the acknowledgement channel. The boundedness property can be even more precise, taking into account the types (sequence numbers) of messages:

$$\begin{aligned}&\text{if } \mathsf{MCOld} \neq \mathsf{MCNew} \text{ then } \mathsf{NbMCOld} \leq \mathsf{MaxRetrans} \wedge \mathsf{NbMCNew} \leq \mathsf{MaxRetrans} + 1\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else } \mathsf{NbMCOld} \leq \mathsf{MaxRetrans} + 1\end{aligned}$$

Stop-and-Wait Property A sent message is received before the next (new) message is sent (i.e. alternating send and receive events.)

No data loss Each original message (or a retransmission) is eventually received, except for the last message in case the original plus all retransmissions were lost, and the maximum number of retransmissions is reached.

No duplication When a duplicate message arrives, it is detected as such and discarded. No duplicate message is mistakenly accepted as a new one.

In-sequence delivery The messages are received in the order they are sent.

Deadlocks When using reliable channels, there should be no deadlock. When using unreliable channels, only expected deadlocks should exist: the maximum number of retransmissions is reached but the sender is stuck waiting for an acknowlegement, and both message and acknowledgement channels are empty:

$$\begin{gathered}\mathsf{retrans} = \mathsf{MaxRetrans}, \mathsf{SState} = 0,\\ \mathsf{MCOld} = \mathsf{MCNew}, \mathsf{NbMCOld} = \mathsf{NbMCNew} = 0,\\ \mathsf{ACOld} = \mathsf{ACNew}, \mathsf{NbACOld} = \mathsf{NbACNew} = 0\end{gathered}$$

5.3 Instrumentation of the model

In order to check several of the properties, some instrumentation of the model is required. We add a variable SRprop, which is set to the sequence number plus 1, when a new message is sent. When an expected message (i.e. not a duplicate) is received, this variable is set to 0. Checking the stop-and-wait property then amounts to verifying that there is no pending new message in the message channel when the sender is ready to send, i.e. no state such that:

$$\mathsf{SRprop} > 0 \wedge \mathsf{SState} = 1$$

When operating over a FIFO medium, because the stop-and-wait property holds (a new message can only be sent if the expected one was received) it follows that there is no loss of data (except possibly the last message as described.)

To verify the no duplication property, we check that there is no state such that the receiver is ready to accept a new message with sequence number other than the most recently sent by the sender, i.e. there is no state such that:

$$\mathsf{SRprop} = \mathsf{MCOld} + 1 \wedge \mathsf{RState} = 1 \wedge \mathsf{NbMCOld} > 0 \wedge \neg(\mathsf{MCOld} = \mathsf{RSeqNb})$$

Effectively, when a duplicate is received, the value of SRprop should be either 0 if no new message has yet been sent by the sender, or corresponds to the sequence number plus 1 of the new message sent, i.e. a different sequence number to the duplicate being received.

Finally, to prove the in-sequence delivery property, we note that variable SRprop contains the number (plus one) of the last new message sent, and that it is not possible to receive an original message with a sequence number different to that most recently sent, i.e.:

$$\neg(\mathsf{SRprop} = \mathsf{MCOld} + 1) \wedge \mathsf{RState} = 1 \wedge \mathsf{NbMCOld} > 0 \wedge \mathsf{MCOld} = \mathsf{RSeqNb}$$

6 Analysis of SWP using FAST

6.1 Introduction to FAST

FAST [3, 4] is a tool dedicated to checking safety properties on counter systems. The main issue addressed by FAST is the *exact* computation of the (infinite) state space. On such a complex problem, although FAST uses a semi-algorithm which is not guaranteed to terminate, experiments with its use on practical examples have been promising [16].

6.1.1 Inputs and Outputs

FAST inputs are in the form of both a *model* and a *strategy* for the analysis. Outputs are messages indicating whether the system satisfies a property or not. The model input format was described in Section 4 where an excerpt of our SWP model was given.

The strategy is the sequence of computations to perform in order to check the validity of the system. The strategy language is a script language which operates on regions (sets of states), transitions and booleans. All the usual operators on sets are available and primitives to compute the reachability set (forward or backward) are provided. Checking a safety property involves declaring the initial states, computing the reachability set A, declaring the property to check (*good states*) B, and testing if $A \subseteq B$.

Here, we show an excerpt of the SWP CS strategy, illustrating FAST strategy input.

```
strategy SWP {
...
Region init := {SState=1 && SSeqNb=0 && Retrans=0 && MCOld=0
&& MCNew=0 && NbMCOld=0 && NbMCNew=0 && ...};

Region reach := post*(init, t, 2);

// Consecutive sequence numbers in Message channel
Region diffminmaxM := {(MCOld=MCNew) || (MCNew=MCOld+1) ||
(MCOld=MaxSeqNb && MCNew=0)};

if (subSet(reach,diffminmaxM)) then
  print("M channel consecutive seq numbers OK");
  else print("M channel consecutive seq numbers NOK");
endif
...
}
```

First, a region init is declared, used to describe the initial states. Then, the set of reachable states reach is computed from init, using forward reachability (function post*). Region diffminmaxM characterises the set of states with consecutive sequence numbers in the message channel. If reach is a subset of diffminmaxM then the consecutive sequence numbers property is satisfied, otherwise it is not. An appropriate message is printed.

6.1.2 Architecture

The FAST computational engine can be used as a standalone application, or with a graphical user interface in a client-server architecture [4]:

- *the server* is the computation engine of FAST. It contains a Presburger library, the acceleration algorithm and the search heuristics;
- *the client* is a front-end which allows interaction with the server through a graphical user interface. This interface facilitates guided editing of models and strategies, with features such as pretty printing and predefined strategies. Once the computation starts, feedback is supplied through different measures and graphs (time elapsed, memory used, number of states, ...).

6.2 Analysis Results

The results obtained by FAST confirm the expected properties from Section 5. They are automatically checked for all values of the MaxRetrans parameter, although FAST did not terminate in a reasonable amount of time when the maximum sequence number was also a parameter. Therefore, we conducted separate runs of FAST with MaxSeqNb fixed, over the range from 1 to 5. The analysis was performed on the lossy channel model as well as on a model with reliable channels (where the loss transitions were removed).

Column *Channels* in Table 1 indicates whether the channel is reliable or lossy. MaxSeqNb gives the values of the variable for the experiment.

The computation is done at a reasonable or even low cost w.r.t. both time and memory usage, as shown by the experimental results in Table 1 for $1 \leq$ MaxSeqNb ≤ 5. The FAST computation is divided into three steps (for technical details, see e.g. [17]):

- transition compositions which take 2 minutes 5 seconds in the lossy case (529 compositions) and 1 minute 36 seconds in the reliable case (289 compositions);
- accelerations computation which takes 31 seconds for 126 cycles in the lossy case and 1 minute 13 seconds for 105 cycles in the reliable case;
- applying the accelerations to construct the state space.

The computation time for compositions and accelerations is exactly the same for all cases, as the same preliminary computations are performed. The differences in time result from the size of the internal representation of each element.

Table 1 gives the total computation times, as well as the peak memory recorded. Column *Nb states* indicates the number of symbolic states at the end of the computation.

Channels	MaxSeqNb	Nb compositions	time (hh:mm:ss)	memory (MB)	Nb states	Nb accelerations	Nb cycles
Reliable	1	289	00:07:34	31	74	64	105
Reliable	2	289	00:36:29	37	167	113	105
Reliable	3	289	00:54:26	44	169	120	105
Reliable	4	289	02:07:14	48	349	123	105
Reliable	5	289	03:00:16	48	360	181	105
Lossy	1	529	00:12:29	19	87	60	126
Lossy	2	529	00:33:52	23	205	132	126
Lossy	3	529	01:28:56	27	199	193	126
Lossy	4	529	03:04:54	38	446	202	126
Lossy	5	529	03:30:21	39	432	233	126

Table 1: Experimental results

7 Conclusions and Future Work

Finite state methods for protocol verification can fail due to state explosion when considering ranges of values for important parameters such as the maximum number of retransmissions or the size of the sequence number space. When considering these parameters, we would like to provide a general result that allows protocol properties to be proved for any value of each parameter. When arbitrary values are considered, we need to generate an infinite number of finite state spaces, one for each value of the parameter. (This is quite different from considering, for example, the specific case of no limit on the number of retransmissions, which gives rise to a single infinite state system.)

This paper has addressed this problem for the stop-and-wait class of protocols, where we modelled the parameters explicitly. We used a recently developed tool called FAST to facilitate parametric verification. FAST allows symbolic state spaces to be generated by taking advantage of encoding arbitrary iterations of sequences of events, known as accelerations. It is based on counter systems, which are automata where states are vectors of (unbounded) integers.

The stop-and-wait protocol (SWP) has two parameters: MaxRetrans representing the maximum number of retransmissions; and MaxSeqNb representing the maximum sequence number that can be used. In previous work [8] we modelled the SWP using Coloured Petri Nets and provided a hand proof that the bound on the number of messages in the FIFO communication channel was 2 MaxRetrans +

1. However, we were only able to prove other properties, such as the stop-and-wait property of alternating sends and receives, for up to 10 bit sequence numbers and with up to 4 retransmissions using automated finite state techniques.

In this paper we have overcome these limitations for the MaxRetrans parameter. Fully automatic proofs have been obtained for channel bounds (confirming the previous hand proofs and including proving that the sum of the messages and acknowledgements in the channels does not exceed 2 MaxRetrans + 1), the stop-and-wait property, that there is no loss of messages (except for the last one when the maximum number of retransmissions is reached), no duplication and that messages are delivered in-sequence. This has been done for $1 \leq$ MaxSeqNb ≤ 5. Unfortunately, FAST does not terminate in a reasonable amount of time when MaxSeqNb is considered as an unbounded parameter, or for values greater than 5. However, we believe this experience will assist us with hand proofs that the results hold for any positive integer value of MaxSeqNb.

Further we have shown how to translate our CPN model into a counter system by using a novel approach to represent a FIFO queue by 4 integer variables. This is valid when the queue can hold only two types of message indicated by their sequence numbers and all messages of the same sequence number are adjacent. This condition is proved using FAST as part of model validation. Some general guidance has also been given for translating CPNs to counter systems.

Future challenges include generalising the method to channels that allow re-ordering of messages and formally incorporating data independence, which has been assumed in our work so far. Other ways of representing queues (perhaps with one integer variable) that are efficient and suit the FAST framework of Presburger arithmetic could be investigated. A more general and formal translation of CPNs into counter systems is also of interest, to allow models that have already been constructed in CPNs to be automatically translated and input to FAST. Automatically translating the properties formulated on the CPN model to those on the counter system and translating the results back is also an interesting issue. We would also like to investigate the use of other tools such as TReX and compare them with FAST.

Acknowledgements

We thank the anonymous referees for their useful comments which helped us to improve our paper.

References

[1] P. Abdulla, A. Annichini, and A. Bouajjani. Symbolic verification of lossy channel systems: Application to the bounded retransmission protocol. In *Proceedings of TACAS'99*, volume 1579 of *LNCS*, pages 208–222. Springer-Verlag, 1999.

[2] P. Aziz Abdulla, A. Collomb-Annichini, A. Bouajjani, and B. Jonsson. Using Forward Reachability Analysis for Verification of Lossy Channel Systems. *Formal Methods in System Design*, 25(1):39–65, 2004.

[3] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In *Proceedings of CAV'2003*, volume 2725 of *LNCS*, pages 118–121. Springer, 2003.

[4] S. Bardin and L. Petrucci. From PNML to counter systems for accelerating Petri nets with FAST. In *Proc. of the Workshop on Interchange Formats for Petri Nets (at ICATPN 2004)*, June 2004.

[5] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. *Communications of the ACM*, 12(5):260–261, May 1969.

[6] B. Bérard and L. Fribourg. Reachability analysis of (timed) Petri nets using real arithmetic. In *Proceedings of CONCUR'99*, volume 1664 of *LNCS*, pages 178–193. Springer, 1999.

[7] J. Billington and G. E. Gallasch. An Investigation of the Properties of Stop-and-Wait Protocols over Channels which can Re-order messages. Technical Report 15, CSEC, University of South Australia, Australia, May 2004.

[8] J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *LNCS*, pages 210–290. Springer-Verlag, 2004.

[9] J. Billington, G.R. Wheeler, and M.C. Wilbur-Ham. PROTEAN: A High-level Petri Net Tool for the Specification and Verification of Communication Protocols. *IEEE Transactions on Software Engineering*, 14(3):301–316, March 1988.

[10] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Proceedings of CAAP'96*, volume 1059 of *LNCS*, pages 30–43. Springer, 1996.

[11] CADP *homepage*. `http://www.inrialpes.fr/vasy/cadp/`.

[12] CCITT. ISDN user-network interface Data link layer specification. Technical report, Draft Recommendation Q.921, Working Party XI/6, Issue 7, Jan. 1984.

[13] DESIGN/CPN *online*. `http://www.daimi.au.dk/designCPN`.

[14] M. Diaz. Modelling and Analysis of Communication and Co-operation Protocols Using Petri Net Based Models. In *Protocol Specification, Testing and Verification*, pages 465–510. North-Holland, 1982.

[15] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proceedings of LICS'99*, pages 352–359. IEEE CS Press, 1999.

[16] FAST *homepage*. `http://www.lsv.ens-cachan.fr/fast/`.

[17] A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *Proceedings of FST&TCS'2002*, volume 2556 of *LNCS*, pages 145–156. Springer, 2002.

[18] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer, second edition, 1997.

[19] J. Leroux. The affine hull of a binary automaton is computable in polynomial time. In *Proceedings of INFINITY'2003*, Electronic Notes in Theor. Comp. Sci. Elsevier Science, 2003.

[20] M. A. Marsan, A. Bianco, L. Ciminiera, R. Sisto, and A. Valenzano. A LOTOS Extension for the Performance Analysis of Distributed Systems. *IEEE Transactions on Networking*, 2(2):151–165, 1994.

[21] W. Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer-Verlag, 1998.

[22] W. Stallings. *Data and Computer Communications*. Prentice Hall, 6th edition, 2000.

[23] L.J. Steggles and P. Kosiuczenko. A Timed Rewriting Logic Semantics for SDL: a case study of the Alternating Bit Protocol. *Electronic Notes in Theoretical Computer Science*, 15, 1998.

[24] I. Suzuki. Formal Analysis of the Alternating Bit Protocol by Temporal Petri Nets. *IEEE Transactions on Software Engineering*, 16(11):1273–1281, 1990.

[25] A. Tanenbaum. *Computer Networks*. Prentice Hall, 4th edition, 2003.

[26] TREX *homepage*. `http://www.liafa.jussieu.fr/~sighirea/trex`.

[27] K. J. Turner (Ed.). *Using Formal Description Techniques: An Introduction to Estelle, Lotos and SDL*. Wiley Series in Communication and Distributed Systems. John Wiley & Sons, 1993.

[28] A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer-Verlag, 1998.

[29] A. Valmari and I. Kokkarinen. Unbounded Verification Results by Finite-State Compositional Techniques: 10^{any} States and Beyond. In *Proceedings of ACSD'98*, pages 75–85. IEEE CS Press, March 1998.

[30] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Proceedings of TACAS'2000*, volume 1785 of *LNCS*, pages 1–19. Springer, 2000.

Validating UML and OCL models in SOCLe by simulation and model-checking [*]

Damien Azambre, Mathieu Bergeron[†] and John Mullins[‡]
Dept. of Computer Engineering, École Polytechnique de Montréal[§]

Abstract

We present a toolset that offers dynamic veri cation of OCL *constraints on* UML *models. Veri cation is a necessary step when designing critical object-oriented software. Our toolset translates a* UML *model into an Abstract State Machine and translates each* OCL *constraint into a set of μ-formulas. Veri cation is done through model-checking every μ-formula against the* ASM*'s behavior.*

1. Introduction

In recent years, the Uni ed Modeling Language (UML) has been accepted as a *de facto* standard for object-oriented software design. The UML notation supports designers by allowing them to express structural and behavioral aspects of their design, mainly through class diagrams and statechart diagrams respectively. Based on mathematical logic, the Object Constraint Language (OCL) is a notation embedded in UML allowing constraint speci cations such as well-formedness conditions (e.g. in the de nition of UML itself) and contracts between parts of the modeled system (e.g. class invariants or method pre- and post-conditions).

Also, used alongside formal method based tools, UML/OCL offers a unique opportunity for developing complex or critical software systems with high quality standards in an industrial context. Such systems require a high level guarantee that they cope with their speci cations from end to end of their development cycle.

SOCLe is a model-checker that uses UML diagrams as a modeling language and extended OCL constraints as a speci cation language. The latter are encoded

[*] The SOCLe project is sponsored by Defence Research and Development Canada (DRDC) (Government of Canada).

[†] Supported by an NSERC scholarship for graduate studies (Government of Canada).

[‡] Corresponding author. Supported by an NSERC individual research grant (Government of Canada).

[§] *Mailing address:* P.O. Box 6079, Stn. Centre-ville, Montreal (Quebec), Canada, H3C 3A7. *E-mail addresses:* {damien.azambre, mathieu-2.bergeron, john.mullins}@polymtl.ca

in propositional μ-calculus whose atomic formulas are OCL expressions, hence enriching contracts with expressive temporal constructs in order to allow software engineers to specify constraints on the temporal evolution of a system structure.

This work is part of a broader project, the SOCLe [1] project, whose aim is building a basis for UML/OCL based methodology and a validation environment for complex systems. The work presented in this paper only addresses the tool-related issues such as: tool architecture, illustration of the basic principles of the semantics de nition, and examples presenting the way the tool applies to speci c models. The semantics itself is presented only to the extent necessary for understanding the paper. For more details on the actual semantics the reader is referred to [2].

The framework includes: An Abstract State Machine (ASM) based semantics of a signi cant fragment of UML including class diagrams, objects diagrams and statechart and supporting the main object-oriented software features such as concurrency, inheritance and object creation; A compiler mapping UML diagrams to an ASM implementing this semantics; A semantics of OCL expressions integrated within the UML semantics for guards, method calls and assignments; An ASM external function implementing this semantics as a recursive function; An ASM simulator computing the behavior of a UML model. We have chosen the formalism of ASM for its expressiveness and its high level of abstraction. A propositional μ-calculus to enrich OCL constraints is integrated along the lines proposed in [4]. It would be unrealistic to expect most developers to acquire an understanding of temporal logic with x points. Also following a Brad eld et al. suggestion [4], we design templates with their own user-friendly syntax to express liveness constraints and we show how the existing OCL contract types (invariants and pre- and post-conditions) may be regarded as such templates.

This paper is structured as follows. Section 2 discusses similar tools found in the literature. Section 3 presents the architecture of our toolset. Sections 4, 5, 6 and 7 sketch the multiple semantics integrated in the framework using a running example. Section 8 completes the example and illustrates the need for extended OCL constraints. Finally, Section 9 presents ongoing improvements to the toolset and conclusions. Note that we assume some familiarity with model-checking, UML and OCL and the ASM formalism. We refer the reader to [15] for an introduction to UML. Details about OCL expressions can be found in [14]. We refer to Gurevich [10] for further details on Abstract State Machines.

2. Related work

Several software architecture integrating model checking tools within UML have been proposed. However, most of them are based on a semantics of UML de ned in terms of the modeling language of some model-checking tool (e.g. [13, 12, 16, 3, 17]).

[1]Secure OCL extensions.

Lilius and Paltor [13] propose vUML, a tool that translates UML statechart diagrams into Promela, the modeling language of the SPIN model-checker. Speci -cations are expressed through LTL (Linear Temporal Logic) formulas transmitted to SPIN. Similarly, Latella et al. [12] and Schäfer et al. [17] give a correct translation from a subset of UML statechart diagrams into Promela. The toolset proposed by Shen et al. [16] maps the static part of the UML model including OCL expressions on object diagram onto ASM and offers static veri cation including: Syntactic correctness according to well-formedness constraints of the UML metamodel; Coherence of the object diagram with respect to the class diagram. However, OCL expressions are not integrated in the UML semantics and are not evaluated as the model evolves. The statecharts, modeling the behavior, are mapped onto SMV. In [3], Bozga et al. propose a model-checking of operational UML models based on a mapping of UML models into a framework of communicating extended automata (in the IF language) for which there exists veri cation tools.

For specifying properties, some approaches opt for the property speci cation language of the model checker itself, e.g. [13, 12, 16]. Other approaches [17] use speci c diagrams e.g. collaboration diagrams, which specify intended sequences of messages between objects, but are not expressive enough to specify more complex properties. In [3], Bozga et al. propose an extension of UML (*observers classes*) expressive enough to express a large class of linear temporal logic. In contrast to these works, our approach is the only one which focusses on OCL and which allows to verify automatically temporal contracts described in extended OCL about the behavior of the system. As such, SOCLe is the rst OCL constraints model-checker.

There are already works to extend OCL with temporal logic in various directions. Temporal semantics for OCL constraints are suggested by Distefano et al. using CTL (Computational Tree Logic) [7] and a subset of OCL. Inheritance is not considered in this approah. In [8], Flake and Mueller present an OCL extension, also based on CTL. This extension concerns system behavior modeled with statecharts but the evolution of attributes is not considered there. Brad eld et al. propose to extend OCL with temporal constructs based on the observational μ-calculus [4]. The idea is to replace atomic properties of these temporal logics by OCL expressions. These expressions are evaluated dynamically as the formula is veri ed. The authors suggest using *templates* with a user-friendly syntax which then have to be translated to observational μ-calculus. However, this framework has never been integrated to a model-checking tool within UML. In [18], an extension of OCL with elements of a bounded linear temporal logic is proposed by Ziemann and Gogolla. The semantics of this extension is given with respect to sequences of states representing the UML model evolution. However, the authors do not discuss how to compute these sequences from the model's behavioral diagrams, contrasting with the approach presented here.

Finally, some tools support OCL expressions and constraints, but with different objectives in mind. The KeY tool [1] translates OCL constraints into Dynamic

Logic predicates (an extension of the Hoare logic). Constraints are to be proved with the help of a theorem-prover on Java CARD programs. The USE tool [9] offers the evaluation of OCL expression and constraints on manually constructed object models and sequence diagrams. The OCLE tool [5] offers static validation of UML models through metamodel-level OCL constraints and code generation of model-level OCL constraints.

3. Toolset architecture

The SOCLe toolset is divided in three main modules: $i)$ an XmiToAsm compiler, $ii)$ a specialized ASM interpreter and $iii)$ a μ-formula model-checker (Fig. 1). The toolset works with UML models expressed in the XML Metadata Interchange format, which is supported by most UML CASE tools.

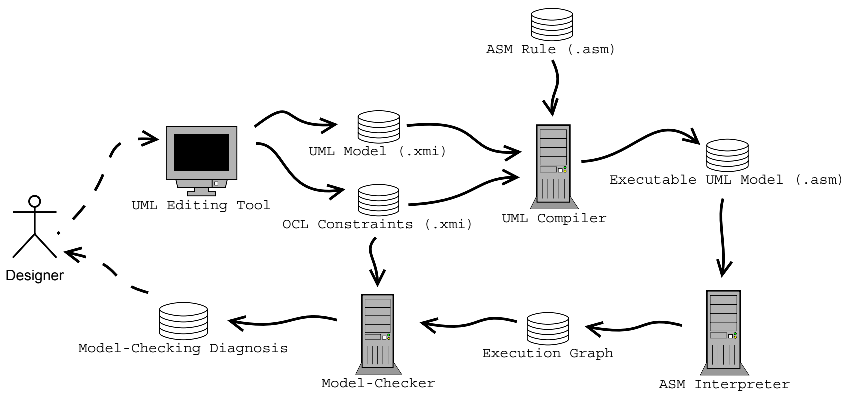


Figure 1. Toolset architecture

The veri cation process has three phases: translating the UML model into an ASM speci cation, executing the model and verifying OCL constraints against the resulting execution graph. The rst two phases rely on the state-based Abstract State Machine formalism. Roughly, an ASM state is a collection of sorts (data domains) and a set of enumerated functions over these sorts. The ASM evolution is speci ed by a transition rule built from predicates, control sub-rules and update sub-rules. Predicates are evaluated according to the current interpretation of the ASM state enumerated functions. Control rules, supporting non-determinism, choose a set of update rules to be applied. Update rules modify the interpretation of the current ASM state functions, hence yielding successor states.

Module $i)$ translates the UML model to an ASM according to a UML model static semantics (Section 4). Basic model elements, such as class or method names, are mapped to sorts. More complex elements, such as method declarations and statechart transitions, are translated into enumerated functions. The object diagram is mapped to a speci c subset of these functions and represents the initial con guration of the UML model (see Subsection 4.3).

Module $ii)$ executes the ASM speci cation. From a con guration, successor con gurations are computed by evaluating an ASM rule capturing the dynamic semantics of UML models (see Section 5). Edges are labeled with statechart transitions red as the UML model evolved.

Module $iii)$ veri es the designer's OCL constraints by translating them into a set of μ-formulas and applying a tableau based μ-calculus veri cation algorithm [6]. The result is transmitted back to the interface through a set of diagnostic les. Section 7 details the model-checking procedure.

The toolset also includes a graphical user interface embedded into ArgoUML, a customizable open-source UML CASE tool developed by Tigris[2]. It allows the designer to visualize veri cation results and inspect the model's execution graph.

4. Static semantics of UML

The UML models supported by the tool must contain exactly one class diagram, one statechart diagram for each class and one object diagram. In this section we illustrate the main features of the static semantics of these three diagrams through a running example, the modeling of a simple object-oriented component acting as a small memory cell.

4.1. Class diagram

Figure 2 presents the supported features of the class diagram. Class *Cell* models a simple memory cell with assignment, retrieval and incrementation. Class *BackupCell* models an extended memory cell with a restore functionality. Notice how class *Client* is tagged with the *thread* stereotype. As a result, a calling stack will be associated with all instances of this class.

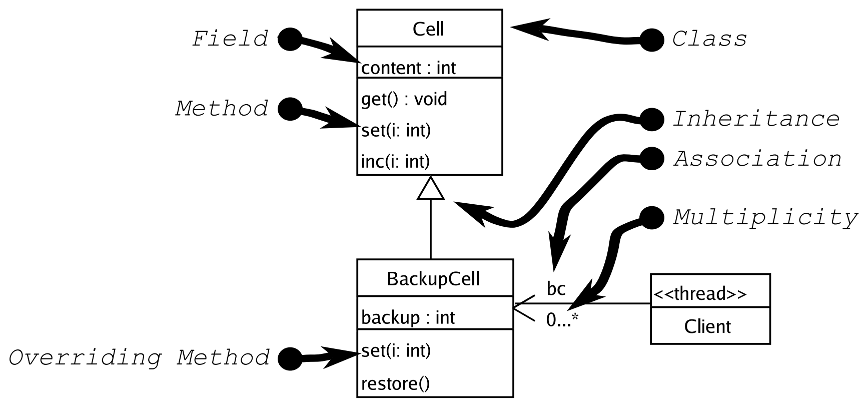


Figure 2. Example of a class diagram

[2]http://argouml.tigris.org/

The rst step to create the `ASM` speci cation is to map class, method and eld names to the following `ASM` sorts (note that an association is mapped to a eld of the owner class):

$$\begin{aligned}
&\textbf{sort } \mathit{ClassName} = \{\mathit{Cell}, \mathit{BackupCell}, \mathit{Client}\}\\
&\textbf{sort } \mathit{FieldName} = \{\mathit{content}, \mathit{backup}, \mathit{bc}\}\\
&\textbf{sort } \mathit{MethName} = \{\mathit{set}, \mathit{get}, \mathit{inc}, \mathit{restore}\}
\end{aligned} \tag{1}$$

Remaining information, like the direct inheritance relation $\sqsubset$, is then extracted and additional functions are elaborated. Here is some examples, which are partially enumerated for space reasons using the $\ldots$ symbol:

$$\begin{aligned}
\textbf{fun} \quad \sqsubseteq \quad &= \mathit{BackupCell} \mapsto \mathit{Cell},\\
& \mathit{Cell} \mapsto \mathit{Cell},\\
& \ldots\\
\textbf{fun} \quad \preceq_o \quad &= \mathit{Cell}/\mathit{set} \mapsto \mathit{Cell}/\mathit{set},\\
& \mathit{BackupCell}/\mathit{set} \mapsto \mathit{Cell}/\mathit{set},\\
& \ldots\\
\textbf{fun } \mathit{lookup} &= \mathit{BackupCell}, \mathit{inc} \mapsto \mathit{Cell},\\
& \mathit{BackupCell}, \mathit{set} \mapsto \mathit{BackupCell},\\
& \ldots
\end{aligned} \tag{2}$$

The $\sqsubseteq$ relation is the inheritance relation. For example, class $\mathit{BackupCell}$ is a subclass of class Cell. The $\preceq_o$ indicates if a method overrides (rede nes) one of its superclass methods. For example, method set of class $\mathit{BackupCell}$ ($\mathit{BackupCell}/\mathit{set}$) overrides method set of class Cell ($\mathit{BackupCell}/\mathit{set}$). These functions are necessary to de ne the important lookup function, which indicates whether re ned or inherited behavior will be executed following a method call. For instance, if method inc is called on an instance of $\mathit{BackupCell}$, the behavior of class Cell will be executed (since that method is not de ned in class $\mathit{BackupCell}$ but inherited from its superclass).

4.2. Statechart diagrams

Similarly, statechart diagrams are mapped to `ASM` sorts and functions. Fig. 3 shows supported features for this diagram. Notice how functionalities of the memory cell are modeled by sub-states specifying the behavior of a method. For example, method inc is modeled in three steps: Transition ct_3 retrieves the current value of eld $\mathit{content}$ by calling method get; Transition ct_4 increments that current value by calling method set; Finally, transition ct_5 waits for method set to return and terminates method inc.

The control o w of a statechart is speci ed by states and transitions. The basic condition for a transition to be red is that its source state is active. The basic response to ring a transition is the activation of its target state. In the case of a

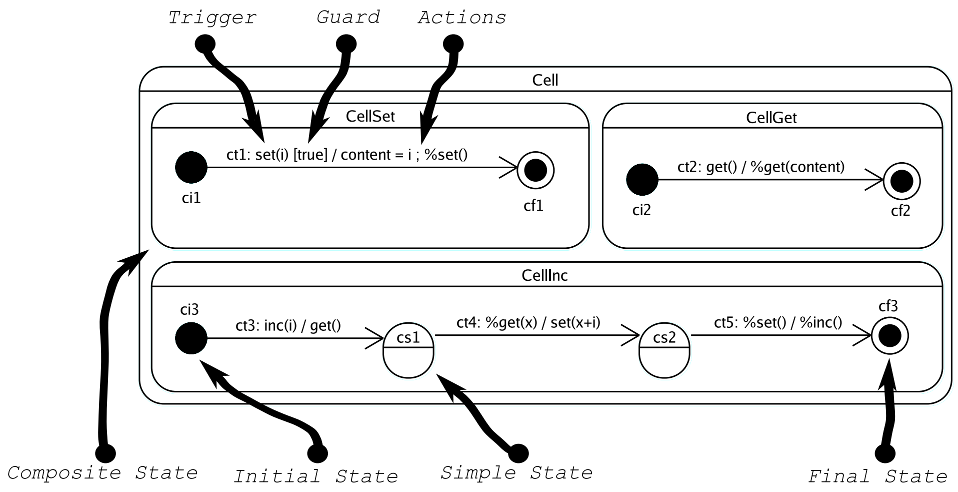


Figure 3. Example of a statechart diagram

composite state, the initial states it encompasses are also activated. This control o w of statecharts is inspired by Harel's statecharts [11] and is statically elaborated and stored in ASM functions. For example, the compiler determines which states are activated and deactivated when ring a transition:

$$
\begin{aligned}
\textbf{fun}\ \ act\ &= ct_1 \mapsto \{\}, \\
&\quad\, ct_3 \mapsto \{cs_1\}, \\
&\quad\, \ldots \\
\textbf{fun}\ deact &= ct_1 \mapsto \{CellSet\}, \\
&\quad\, ct_3 \mapsto \{ci_3\}, \\
&\quad\, \ldots
\end{aligned}
\tag{3}
$$

In addition, transitions are labeled with a trigger, a guard and a list of actions. Triggers refer to signals (atomic events), method calls or method returns. For example, the actions of a transition labeled with trigger inc will model that method's instructions. Guards are boolean OCL expressions. OCL expressions are presented in Section 6. The toolset supports the following actions: method call/return, eld assignment, object creation/deletion, and signal emission. Actions are speci ed in part with OCL expressions, which enables the designer to model high-level behavior by using non-determinism. For instance, in a method call action, an OCL expression speci es a collection of possible receiver objects from which the actual receiver is chosen non-deterministically.

Finally, statechart compilation includes a fair amount of static veri cation: i) statecharts are inspected to insure they are well-formed, ii) OCL expressions are type-checked to insure that guards are boolean expressions, that parameters of method calls are well-typed, etc. iii) triggers and actions are analyzed to insure consistency with the class diagram methods and elds declarations.

4.3. Object diagram

The object diagram is mapped to ASM sorts and functions that hold the UML model con guration. Figure 4 shows such a diagram with all features covered by the toolset. It models a simple con guration in which a client accesses two memory cells.

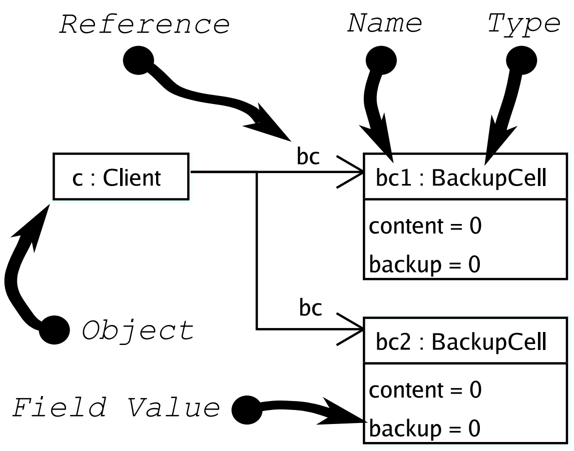


Figure 4. Example of an object diagram

ASM functions as, $class$, $heap$ and st respectively hold active states, objects type and eld environment, and calling stacks (one for each thread; in this case only object c is a thread):

$$
\begin{array}{lll}
\textbf{fun}\ \ as & = bc_1 & \mapsto \{ci_1, ci_2, \ldots\},\\
 & \ \ c & \mapsto \{cli_1\},\\
 & \ \ \ldots & \\
 & & \\
\textbf{fun}\ class & = bc_1 & \mapsto BackupCell,\\
 & \ \ c & \mapsto Client,\\
 & \ \ \ldots & \\
 & & \\
\textbf{fun}\ heap & = bc_1, content & \mapsto 0,\\
 & \ \ bc_1, backup & \mapsto 0,\\
 & \ \ \ldots & \\
 & & \\
\textbf{fun}\ \ st & = c & \mapsto \langle (run, \emptyset, \bot, c)\rangle
\end{array}
\tag{4}
$$

Notice how the calling stack of object c contains method run in the initial con guration to insure that the thread is active.

5. Dynamic semantics of UML

The ASM transition rule capturing UML models dynamic semantics is roughly structured as follows: $i)$ choose the current thread, $ii)$ select the current object and current statechart, $iii)$ choose one of the enabled transition and $iv)$ re the transition.

Subrules $i)$ and $iii)$ use a non-deterministic choice to model both thread-level and statechart-level concurrency. The former models a simple thread scheduler. The current object $\tilde{o}$ is selected from an active thread's calling stack, i.e. $\tilde{o}$ is currently executing a method. The latter computes transition interleaving of a statechart's concurrent regions.

In sub-rule $ii)$, the current statechart is either the statechart of the current object's class or the statechart of one of its superclass if inherited behavior is to be executed (this is decided according to the *lookup* function of Eq. 2). This mechanism captures behavioral inheritance, an important feature of object-orientation.

Sub-rule $iii)$ dynamically determines whether a transition is enabled. The basic condition that the source state is active is checked against the current value of function *as* (Eq. 4). Moreover, a transition is enabled if $a)$ its trigger corresponds to an active event, $b)$ its guard is satis ed and $c)$ all of its actions can be red. For example, an assignment action will not be red if it violates the multiplicity requirement (see Fig. 2).

In sub-rule $iv)$, the selected transition $\tilde{t}$ is red. The basic effect (i.e. deactivating and activating states) is captured by updating function *as* using the statically elaborated functions *act* and *deact* (Eq. 3):

$$as(\tilde{o}) := (as(\tilde{o}) \setminus deact(\tilde{t})) \cup act(\tilde{t}) \tag{5}$$

Then, the transition's action list is iterated and every action is red. Object are created by using the sort extensions mechanism of the ASM formalism and by updating function *heap* (Eq. 4) accordingly. For example, if an object creation action a of the form "**new** f" is red by the current object $\tilde{o}$, the following ASM sub-rule updates the UML model con guration:

$$\begin{array}{l} \mathbf{extend}\ \mathit{Objects}\ \mathbf{with}\ x\ \mathbf{do} \\ \quad \mathit{heap}(\tilde{o}, f) := x :: \mathit{heap}(\tilde{o}, f) \end{array} \tag{6}$$

The assignment action uses the OCL expression evaluation function and updates function *heap* (Eq. 4) accordingly. For example, if an assignment action a of the form "$\mathit{content} \ := \ \mathbf{self}.\mathit{content} \ + \ i$" is red by the current object $\tilde{o}$, the following ASM sub-rule updates the UML model con guration:

$$\mathit{heap}(\tilde{o}, \mathit{content}) \ := \ [\![a.e]\!]_\rho \tag{7}$$

It uses function $[\![\ ]\!]_\rho$ to evaluate $a.e$, the OCL expression of the assignment action (in this case "$\mathbf{self}.content\ +\ i$") and updates function *heap* (Eq. 4). Function $[\![\ ]\!]_\rho$ evaluates an OCL expression by recursively evaluating its sub-expressions, with respect to the current UML con guration and a variable assignment ρ. The environment always maps the variable **self** to the current object $\tilde{o}$. In addition here, it maps variable i to the value of the formal parameter of method *inc* as indicated on the calling stack. As the function is external, it uses ASM functions to access the current UML model con guration but is not enumerated in the ASM state. It implements an OCL expression semantics presented in Section 6.

6. OCL expressions

The syntax of an OCL expression e is given by Figure 5. It includes a signi cant fragment of OCL expressions de ned in [14].

$$\begin{array}{rcl} e & ::= & v \mid x \mid \triangle\, e \mid e \triangle e \mid e\ .\ f \mid \\ & & e_1 \longrightarrow \mathbf{iterate}\ \{x_1\ ;\ x_2 = e_2 \mid e_3\ \} \end{array}$$

Figure 5. OCL expressions syntax

Symbols v and x denote values and variables respectively. Values include booleans, integers, object names and lists. As a simpli cation, booleans are not considered as three-valued and other OCL collections, such as sets and bags (multi-sets), are dropped. Construct $\triangle\ e$ (resp. $e\ \triangle\ e$) stands for the application of a any usual unary (resp. binary) operator on booleans, integers and lists. Construct $e\ .\ f$ returns the value that eld f takes in the object denoted by e. Formally, the semantics of these constructs is de ned as follows:

$$\begin{array}{lcl} [\![v]\!]_\rho & = & v \\ [\![x]\!]_\rho & = & \rho(x) \\ [\![\triangle\ e_1]\!]_\rho & = & \triangle\ [\![e_1]\!]_\rho \\ [\![e_1 \triangle e_2]\!]_\rho & = & [\![e_1]\!]_\rho \triangle [\![e_2]\!]_\rho \\ [\![e_1\ .\ f]\!]_\rho & = & heap([\![e_1]\!]_\rho, f) \end{array} \quad (8)$$

The application $\rho(x)$ retrieves the value of variable x in the variable environment. The application $heap([\![e_1]\!]_\rho, f)$ fetches the value of eld f according to the current interpretation of the ASM function *heap* (Eq. 4).

The **iterate** construct is the OCL main collection operator. It lets variable x_1 iterate through values of the collection denoted by e_1, stores successive values of e_3 in variable x_2 (which rst evaluates to e_2) and returns the nal value of x_2. Formally, the semantics of this construct is de ned as follows:

$$
\begin{array}{l}
[\![e_1 \rightarrow \mathbf{iterate}\{x_1; x_2 = e_2 \mid e_3\}]\!]_\rho \;\; = \\
\\
\textbf{let } v_1 = [\![e_1]\!]_\rho \textbf{ in} \\
\textbf{let } v_2 = [\![e_2]\!]_\rho \textbf{ in} \\
\textbf{case } v_1 \textbf{ of} \\
\quad \langle\rangle : \qquad\quad v_2 \\
\quad \langle l_1 \rangle : \qquad\; [\![e_3]\!]_{\rho[x_1 \mapsto l_1, x_2 \mapsto v_2]} \\
\\
\quad \langle l_1, l_2, \ldots \rangle : \textbf{let } v_3 = [\![e_3]\!]_{\rho[x_1 \mapsto l_1, x_2 \mapsto v_2]} \textbf{ in} \\
\qquad\qquad\qquad [\![\langle l_2, \ldots \rangle \rightarrow \mathbf{iterate}\{x_1; x_2 = v_3 \mid e_3\}]\!]_\rho
\end{array}
\tag{9}
$$

In the rst case, the list to iterate is empty and the construct takes the value of expression e_2. In the second case, the list contains only one value and the construct takes the value of expression e_3, which may refer to variable x_1 or x_2. Consequently, the value of these variables is updated in the variable environment. In the third case, the rst value of the list (l_1) is removed and the evaluation process is iterated on the remaining list. The *iterate* construct is quite expressive and is used to encode additional collection operators, which are supported by the SOCLe toolset. Figure 6 illustrates some of them.

$$
\begin{array}{lcl}
e_1 \rightarrow \mathbf{size} & \equiv & e_1 \rightarrow \mathbf{iterate}(v_1;\ v_2 = 0 \mid v_2 + 1) \\
e_1 \rightarrow \mathbf{forall}\{v_1 \mid e_2\} & \equiv & e_1 \rightarrow \mathbf{iterate}(v_1;\ v_2 = \mathbf{true} \mid v_2\ \wedge\ e_2) \\
e_1 \rightarrow \mathbf{exists}\{v_1 \mid e_2\} & \equiv & e_1 \rightarrow \mathbf{iterate}(v_1;\ v_2 = \mathbf{false} \mid v_2\ \vee\ e_2)
\end{array}
$$

Figure 6. Collection operators

7. OCL constraints

We give a propositional μ-calculus semantics to OCL constraints. Expressions are evaluated during execution graph computation. Atomic properties of a μ-formula are replaced with predicates over a UML con guration. These predicates return information about evaluated OCL expressions and method instances. We illustrate the approach using the OCL constraint of Fig. 7.

$$
\begin{array}{ll}
\textbf{context:}\ \ \mathit{BackupCell} :: \mathit{inc}(i : \mathit{int}) & \\
\quad \textbf{pre:}\ \ \mathit{true} & (e_1) \\
\quad \textbf{post:} \mathit{self.backup} = (\mathit{self.content})@\mathit{pre} & (e_2)
\end{array}
$$

Figure 7. OCL **constraint** c_1

By virtue of its explicit context, the constraint applies to every instance of method inc called on an instance of class $BackupCell$. If the pre-condition (expression e_1) is satis ed, the method must return in a con guration in which the post-condition (expression e_2) holds. The special OCL operator $@pre$ returns the value that an expression had prior to the method call.

In the UML model con guration, speci c ASM functions support the veri cation of a constraint. First of all, OCL expressions are named (here e_1 and e_2). Then, function $meth$ is used to dynamically name method instances (an external function is used to avoid name clashes):

$$\begin{array}{l} \textbf{fun } meth = (bc_1, inc) \mapsto \{inc_1, inc_2, \ldots\}, \\ \quad \ldots \end{array} \tag{10}$$

Then, function *exp* (Eq. 12) holds the evaluation or partial evaluation of constraint-related OCL expressions. The need for partial evaluation arises from the $@pre$ operator. Similarly to OCL expression evaluation, partial evaluation is done by an external and recursive function, namely $\{\!\{\ \}\!\}_p$. Before a method call, function $\{\!\{\ \}\!\}_p$ is used and the $@pre$ expression is evaluated, but the remainder of the expression is left unevaluated:

$$\begin{array}{ll} \textbf{fun } exp = & (inc_1, e_1) \mapsto true, \\ & (inc_1, e_2) \mapsto self.backup = 3, \\ & \ldots \end{array} \tag{11}$$

After the method call, the remainder of the expression is evaluated:

$$\begin{array}{ll} \textbf{fun } exp = & (inc_1, e_1) \mapsto true, \\ & (inc_1, e_2) \mapsto false, \\ & \ldots \end{array} \tag{12}$$

Finally, functions now and $post$ mark the initial con guration of a method instance and the several possible con gurations following its return.

$$\begin{array}{ll} \textbf{fun } now = & inc_1 \mapsto true, \\ & \ldots \\ \textbf{fun } post = & inc_1 \mapsto false, \\ & \ldots \end{array} \tag{13}$$

During the ASM execution, an additional sub-rule updates the aforementioned functions. Figure 8 summarizes information added to UML con gurations in order to verify OCL constraints.

The next step is to derive a μ-formula from the constraint. Figure 9 gives the syntax of any μ-formula ϕ.

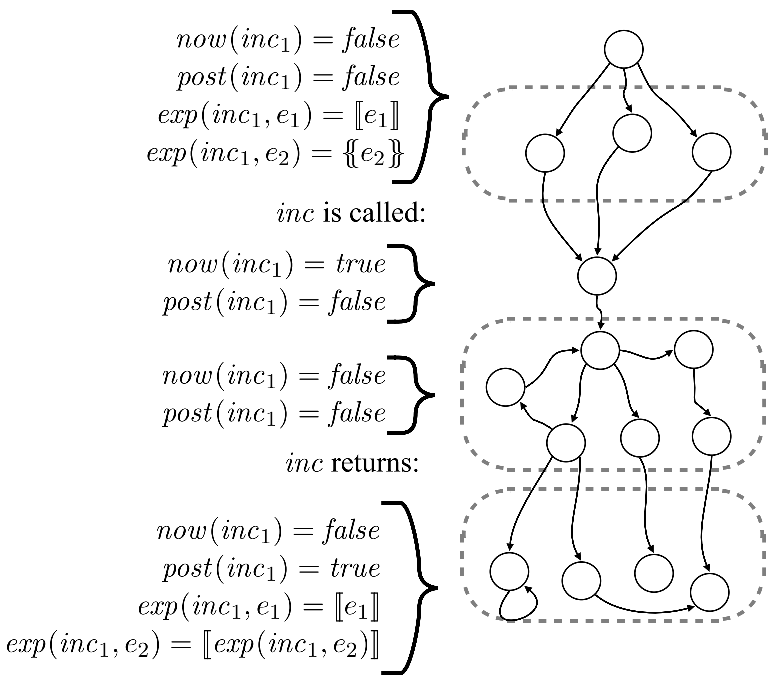


Figure 8. Constraint speci c con guration

$$\begin{aligned} \phi \quad ::= \quad & \Phi \mid X \mid \neg\phi \mid \phi \vee \phi \mid \langle\rangle\phi \mid []\phi \mid \\ & \phi \wedge \phi \mid \phi \Rightarrow \phi \mid \mu X.\phi \mid \nu X.\ \phi \end{aligned}$$

Figure 9. μ-calculus syntax

The symbol Φ denotes a predicate over a `UML` con guration, such as "$exp(inc_1, e_2)$". The symbol X is a μ-variable. The construct $\langle\rangle\phi$ means: ϕ holds in some successor con guration. The construct $[]\phi$ means: ϕ holds in every successor con guration. Intuitively, $\mu X.\phi$ is considered as nitely iterating ϕ and $\nu X.\phi$ as in nitely iterating ϕ. We refer the reader to [6] for a formal semantics of μ-formulas. In order to give a simpli ed account of the implemented μ-formula, we assume a backward existential modality noted $\overleftarrow{\langle\rangle}\,\phi$, which means ϕ holds in some predecessor con guration.

Continuing our example, the formula ϕ_{c_1} below (Eq. 14) is derived from constraint c_1 (Fig. 7).

$$\phi_{c_1} \;\equiv\; \nu X.\ []X \wedge (\phi_1 \Rightarrow \mu Y.\ []Y \vee \phi_2) \tag{14}$$

The μ-formula states that the following is always true: if a method is called with a valid pre-condition (ϕ_1), it will eventually return and satisfy its post-condition (ϕ_2). Equation 15 below details sub-formulas ϕ_1 and ϕ_2. Notice how the free variable m is used as a place-holder for a method instance name.

$$\begin{aligned} \phi_1 &\equiv\ now(m) \wedge \overleftarrow{\langle\rangle}\, exp(m, e_1) \\ \phi_2 &\equiv\ post(m) \wedge exp(m, e_2) \end{aligned} \tag{15}$$

Predicate "$now(m)$" is true in the initial con guration of method instance m. The pre-condition is satis ed if there exists at least one predecessor of that con guration in which the predicate "$exp(m, e_1)$" is true, i.e. in which expression e_1 holds. Formula ϕ_2 is similar and relates to the post-condition.

The last step to verify constraint c_1 is to generate a μ-formula for each instance of method inc. In each μ-formula, the free variable m is replaced with the appropriate method instance name. Formally, a UML model with execution graph Θ satis es constraint c_1 if:

$$\begin{aligned} &\forall\, o\ .\ class(o) \sqsubseteq BackupCell : \\ &\quad \forall\, m'\ .\ m' \in meth(o, inc)\ : \\ &\qquad \Theta \models \Phi_{c_1}[m \mapsto m'] \end{aligned} \tag{16}$$

8. Extending OCL contracts: An example

Recall the class diagram of Fig. 2. As mentioned, class $BackupCell$ models an extended memory cell by adding a restore functionality. To do so, the behavior of method set is rede ned in the class statechart (Fig. 10).

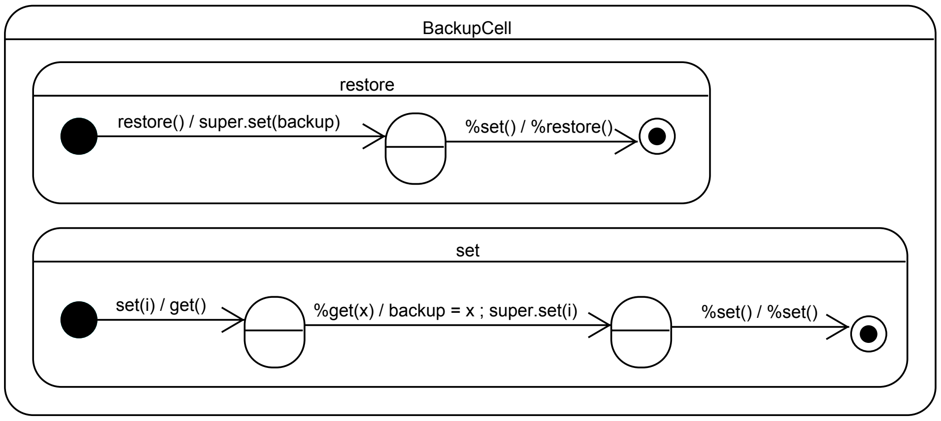


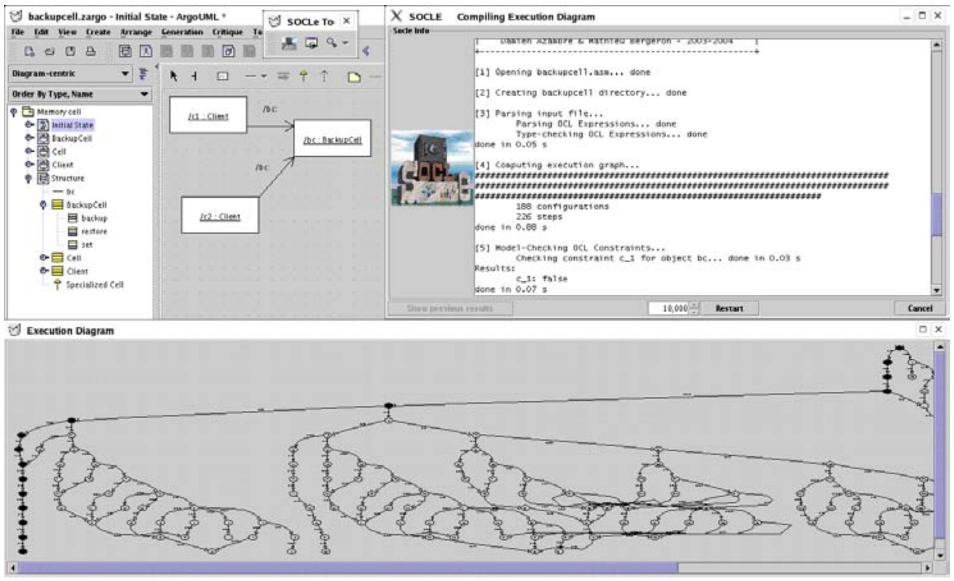
Figure 10. $BackupCell$ **statechart**

First, the value of eld $content$ is copied to eld $backup$, then the eld $content$ is set by calling method set of the superclass. When method inc is called on an instance of class $BackupCell$, $Cell$'s statechart executes the inherited behavior. According to the inheritance semantics, the overridden method set is called to update the value of eld $content$, hence correctly executing the backup copy.

Even though method inc is not rede ned in class $BackupCell$, it exhibits a re ned behavior.

In a rst attempt to validate the re ned method inc's behavior, the designer may use constraint c_1 of Section 7 (Fig. 7). It states that upon return of any instance of method inc (called on an instance of class $BackupCell$), eld $backup$ must contain the value that eld $content$ had prior to the method call.

If the constraint is veri ed on a model that uses Fig. 4 as its initial con guration, the SOCLe toolset diagnoses that the constraint is satis ed. The designer concludes that inheritance was used correctly. Things get more complicated, however, if the initial con guration allows two instances of class $Client$ to concurrently call methods of the same memory cell. When such an initial con guration is used, the SOCLe toolset reports a faulty execution sequence (Fig. 11).



```
[1] Opening backupcell.asm... done

[2] Creating backupcell directory... done

[3] Parsing input file...
        Parsing OCL Expressions... done
        Type-checking OCL Expressions... done
done in 0.05 s

[4] Computing execution graph...
################################################################################
################################################################################
###################################################################
        188 configurations
        226 steps
done in 0.88 s

[5] Model-Checking OCL Constraints...
        Checking constraint c_1 for object bc... done in 0.03 s
Results:
        c_1: false
done in 0.07 s
```

Figure 11. The SOCLe toolset diagnosis

The toolset displays the entire execution graph of the model and highlights the faulty execution sequence. Inspection of the faulty sequence reveals that, as expected, the constraint is violated when concurrent calls are made to method inc (Fig. 12).

The rst instance of method inc is called and res its rst transition. The second instance of method inc is called and returns before the rst instance can execute any other transition. After the second instance has returned, the rst resumes and nishes its job. The constraint is violated (for the rst instance of method inc) as eld $backup$ does not hold the value eld $content$ had prior to the call. Notice, however, that eld $backup$ correctly holds the last value of eld $content$. The designer concludes that a pure post-condition is not suitable to express the

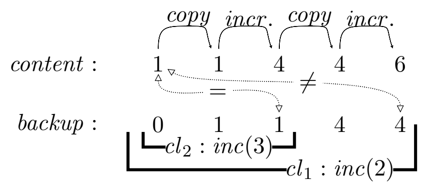


Figure 12. Counter-example for constraint c_1

desired behavior.

Fortunately, our approach allows her to use more intricate constraints, which are supported by the expressive power of the μ-calculus. The designer reformulates her constraint (Fig. 13).

$$\begin{array}{ll}\textbf{context:} & BackupCell :: inc(i : int) \\ \quad\textbf{post:} & self.content = (self.content)@post \\ \quad\textbf{back to:} & self.content = (self.backup)@post\end{array}$$

Figure 13. `OCL` **constraint c_2**

It states that from the post-condition, the expression "$content = content@post$" must hold until (backwards in time) the expression "$content = backup@post$" eventually holds. Although more intricate, it correctly captures the idea that eld *backup* must hold the last value of eld *content*. The designer uses the post-condition of method *inc* as an indicator that the value of eld *content* has changed. A similar constraint should also be formulated for method *set*. Perhaps not surprisingly, that constraint is also violated when concurrent calls are made to the method *inc* (Fig. 14).

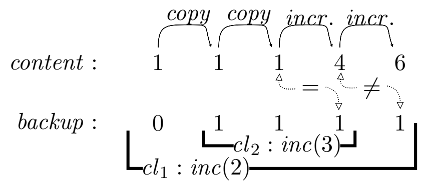


Figure 14. Counter-example for constraint c_2

This time, the rst instance of method *inc* executes the copy before the second instance executes in its entirety. When the rst instance resumes, it executes the incrementation. Again, the constraint is violated for the rst instance.

Finally, the designer concludes that copy and incrementation actions must not be interrupted. This is modeled in `UML` by using signals preventing critical sections from being interrupted. Once corrected, the model satis es the second `OCL` constraint even when concurrent calls to method inc occur.

Some remarks follows from this simple example. Even with a simple model, behavioral inheritance is dif cult to use and needs careful modeling and veri cation. The SOCLe toolset has helped the designer to clearly express the intended behavior of her model. Furthermore, analysis of faulty execution sequences helped pinpoint the source of error due to concurrency interleaving. We believe that early formulation of the concurrency scheme for software can greatly aid in exposing these errors that are dif cult to detect using conventional testing techniques.

9. Conclusion and future works

In this paper, we have presented a toolset that enforces static veri cations on a `UML` model and mechanically veri es `OCL` constraints. Our toolset relies on: An `ASM` based semantics of `UML` models; A recursive `OCL` expression evaluation function de ned with respect to `UML` con gurations; A propositional μ-calculus built from boolean `OCL` expressions that acts as an `OCL` constraint language; And a tableau based veri cation algorithm of extended `OCL` constraints. We have illustrated how the `ASM` semantics captures complex features of `UML` like concurrency, inheritance and object creation, and developed an `ASM` interpreter specialized for this semantics. We have also de ned and illustrated the formal semantics of a fragment of the `OCL` expressions and extended `OCL` constraints as implemented in the toolset. We have nally motivated, through an example, the need for specifying and verifying extended contracts and how the toolset supports such veri cation.

Three major improvements of the toolset are currently underway: $i)$ model reduction and $ii)$ on-the- y model-checking of `OCL` constraints and $iii)$ symbolic model-checking. Improvement $i)$ consists of translating the `UML` model into an intermediate control o w language before using the `ASM` formalism. The intermediary representation facilitates the implementation of various static analysis techniques such as slicing and code generation. Improvement $ii)$ utilizes a local model-checking algorithm to drive the computation of a model's execution graph. Memory requirements are greatly reduced as some constraints can be falsi ed/v eri ed even if the execution graph is only partially computed. Improvement $iii)$ consists to use compact state sets representation combined with veri cation techniques allowing to handle directly these sets from their symbolic representation. This should result in a signi cant breakthrough because it will allow systems with much larger state spaces to be veri ed.

References

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. Technical Report 2003-05, Department of Computing Science, Chalmers University of Technology and Göteborg University, March 2003.

[2] Mathieu Bergeron and John Mullins. Model-checking UML Designs Using a Temporal Extension of the Object Constraint Language. Technical Report 1/04, École Polytechnique de Montréal, 2004. http://www.polymtl.ca/crac/socle/publications.html.

[3] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. Tools and applications II: The if toolset. In Flavio Corradinni and Marco Bernanrdo, editors, *Proceedings of SFM'04*, volume 3185 of *LNCS*, Bertinoro, Italy, 2004. Springer.

[4] Julian Brad eld, Juliana Küster Filipe, and Perdita Stevens. Enriching OCL using observational mu-calculus. *Lecture Notes in Computer Science*, 2306:203–??, 2002.

[5] Dan Chiorean, Mihai Pasca, Adrian Carcu, Cristian Botiza, and Sorin Moldovan. Ensuring UML Models Consistency Using The OCL Environment. In *UML 2003 - OCL Workshop*, October 2003.

[6] R. Cleaveland. Tableaux-Based Model Checking in the Propositional μ-calculus. *Acta Informatica*, 27:725–747, 1990.

[7] Dino Distefano, Joost-Pieter Katoen, and Rensink Rensink. On a temporal logic for object-based systems. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2000, September, 2000, Stanford, California, USA*. Kluwer Academic Publishers, 2000.

[8] Stephan Flake and Wolfgang Mueller. An OCL extension for real-time constraints. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 150–171. Springer, 2002.

[9] Martin Gogolla, Mark Richters, and Jörn Bohling. Tool Support for Validating UML and OCL Models Through Automatic Snapshot Generation. In *SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 248–257. South African Institute for Computer Scientists and Information Technologists, 2003.

[10] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Speci cation and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[11] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[12] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic veri cation of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *The International Journal of Formal Methods*, 11(6):637–664, 1999.

[13] Johan Lilius and Ivan Porres Paltor. vUML: a tool for verifying UML models. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 255–258. IEEE Computer Society, 1999.

[14] OMG. Response to the UML 2.0 OCL RfP (ad/2000-09-03). Technical Report ad/2002-05-09, 2002.

[15] J. Rumbaugh, I. Jacobson, and G. Booch. *Uni ed Modeling Language Reference Manual*. Addison-Wesley, 1998.

[16] Wuwei Shen, Kevin Compton, and James K. Huggins. A toolset for supporting UML static and dynamic model checking. In *26th IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 147–152, Oxford, England, August 2002. IEEE Computer Society.

[17] Stephan Merz Timm Schäfer, Alexander Knapp. Model Checking UML State Machines and Collaborations. In *CAV 2001 Workshop on Software Model CheckingAlgebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Paris, France*, volume 55 (3) of *ENTCS*, 2001.

[18] Paul Ziemann and Martin Gogolla. An OCL extension for formulating temporal constraints. Technical Report 1/03, Universität Bremen, 2003.

Deriving Software Product Line's Architectural Requirements from Use Cases: an Experimental Approach

Alexandre Bragança[1] and Ricardo J. Machado[2]

[1] Dep. Eng. Informática, ISEP, IPP, Porto, Portugal,
alex@dei.isep.ipp.pt
[2] Dep. Sistemas de Informação, Universidade do Minho, Guimarães, Portugal,
rmac@dsi.uminho.pt

Abstract

One of the most important artifacts of a product line is the product line architecture. In this paper we present an approach for deriving a product line's architecture from the requirements of the product line. This approach is based on a transformational technique that has been developed and applied to obtain system architectures from requirements specified as UML use cases. In this paper we evaluate if such a technique can be applied to product lines and, if so, what adaptations are required. For presentation purposes we use the public available IESE report of the *GoPhone* product line that uses the UML modeling language.

1. Introduction

One of the most important artifacts of a product line is the product line architecture. A product line architecture is the basis for the derivation of the architectures of the members of the product line and also of the development of reusable product line components. As such, the product line architecture must encompass all the actual members of the product line as well as future members. This makes it a crucial artifact of the product line engineering process.

As for single systems development, the reference architecture for a product line is basically obtained from requirements. UML use cases are a widely adopted technique for functional requirements modeling. They are used with this perspective in single system development and also in product line approaches [1]. In a product line approach requirements result from domain analysis. The domain analysis phase of product line engineering may involve several specific activities, besides functional requirements modeling, such as product line scoping and product portfolio definition.

The scoping activity aims at defining the products that the product line may include. In order to do so it is necessary to identify what is the domain of the product line and what are external and sub-domains. The result is usually a diagram representing the relations between domains.

The product portfolio aims at identifying the exact members of the product line, its characteristics and the timing for its development. To differentiate between products of the product line it is necessary to identify its features. Some features are common to several members of the product line while others are not. Feature diagrams are usually adopted for this purpose [2].

The two major techniques for dealing with requirements in a product line approach are use cases and feature models. They can be used together: the use case model is user oriented while the feature model is *reuser* oriented [3]. In this way, use cases focuses on requirements elicitation (what functionality should by provided by the product line), while features address better the functionality that can be *composed* for the members of the domain.

Regarding the reference architecture of a product line, use case models are the driving force that guides its development. Nevertheless, there are not documented processes in the product line area to help in the transition from use case requirements to high-level reference architectures. For instance, RSEB[4] (Reuse-driven Software Engineering Business) proposes that each use case gives origin to three kinds of objects, following the boundary-control-data pattern. But this is still just the starting point of the process. Other methods, like PuLSE [5], simply provide a framework for guiding the design and evaluation of the product line architecture.

In this context, we find that the derivation of a high-level architecture from the requirements of a product line is still a topic of the product line engineering process that needs further research. In this paper we address this problem. Our approach is based on a proven technique that has been used for the derivation of single system architectures from requirements modeled as UML use cases [6]. The 4SRS (4-Step Rule Set) technique applies transformational steps in order to derive a high-level architecture (system-level object model) from the requirements of a system. In order to use this technique in the product line context, adaptations are needed. For instance, the technique has to address the variability concept that is essential to product lines. In order to best evaluate our approach we use, along the paper, the publicly available IESE *GoPhone* product line technical report [7]. This technical report presents a mobile phone product line engineered using PuLSE and KobrA. KobrA is an object-oriented customization of the PuLSE method [8].

The remainder of this paper is structured as follows. Section 2 discusses product line requirements modeling based on use cases. Section 3 describes the application of the 4SRS to derive an architecture for the *GoPhone* product line. It also discusses the modifications required to adapt 4SRS to product lines. The 4SRS resulting logical architecture is presented in Section 4 and a comparison is made with the architecture of the original *GoPhone* from the IESE report. This Section also addresses the instantiation process of architectures for members of the product line and the role of feature diagrams. Section 5 concludes the paper.

2. Requirements Modeling

Functional requirements of product lines can be modeled by use cases. Use case modeling in a product line must capture the requirements for all the possible members of the product line. As such, when adopting use cases to model the requirements of a product line, the major issue is the representation of variability. This means that each use case can vary, depending on the functional requirements of the members of the product line.

Variability is usually modeled using the concept of *variation points*. These variation points identify locations where variation will occur. In use cases, variation points can be expressed in different ways: includes relationship, extension points and use case parameters. To our knowledge, extension points are the more common way of expressing variability in use cases.

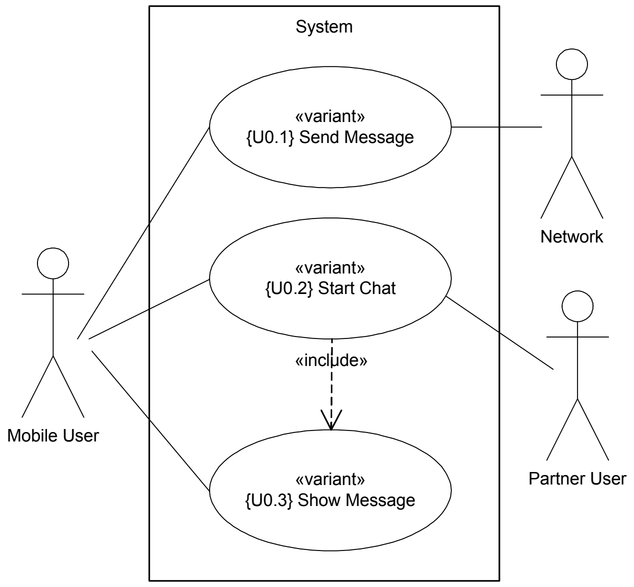


Fig. 1. Use case diagram depicting the main functionality of the messaging domain (Based on the IESE's GoPhone Technical Report [7])

Variability can also be modeled in use case diagrams by using stereotypes to mark use cases. For instance, Gomaa proposes three stereotypes to classify use cases regarding variability: «mandatory», «optional» and «alternative» [1].

Send Message

1. The user chooses the menu-item to send a message.
2. The user chooses the menu-item to start a new message.
3. Are there various message types?
<OPT> The system asks the user which kind of message he wants to send (Go Phone S, M, L, XL, Elegance, Com, Smart)
4. The system switches to a text editor.
5. The user enters the text message.
6. Is T9 supported?
<ALT 1> If T9 is activated, the system compares the entered word with the dictionary. (Go Phone XS, S, M, L, XL, Elegance)
7. Which kind of objects can be inserted into a message?
<ALT 1> The user can insert a picture into the message (Go Phone S, M, L, XL)
<ALT 2> The user can insert a picture or a drafted text-element into the message. (Go Phone Elegance, Com, Smart)
<ALT 3> Ø (Go Phone XS)
8. Which kind of objects can be attached to a message?
<ALT 1> The user can attach files, business cards, calendar entries or sounds to the message. (Go Phone Smart)
<ALT 2> The user can attach business cards or calendar entries to the message.(Go Phone S, M, L, XL, Elegance, Com)
<ALT 3> Ø (Go Phone XS)
9. The user chooses the menu-item to send the message.
10. The system asks the user for a recipient.
11. Which kind of message will be sent?
<ALT 1> The user types the phone number or chooses the recipient from the addressbook.(Go Phone XS, S, M, L, XL, Elegance)
<ALT 2> In case of a basic or extended SMS, the user types the phone number or chooses the recipient from the addressbook. In case of an email, the user types the email-address or chooses the recipient from the addressbook. (Go Phone Com, Smart)
12. The system connects to the network and sends the message, then the system waits for an acknowledgement.
13. The network sends an acknowledgement to the system.
14. The system shows an acknowledgement to the user that the message was successfully sent.
15. Is a sent message directly saved in the sent-message folder?
<ALT 1> The system asks the user if the message should be saved. If it should be saved, the system saves the message in the 'sent-message' folder (Go Phone XS, S, M, L, XL, Elegance)
<ALT 2> The system saves the message in the 'sent-message' folder.
(Go Phone Com, Smart)
16. The system switches to the main menu.

Fig. 2. Description of the use case *Send Message* (Based on the IESE's GoPhone Technical Report [7])

In GoPhone, a variant use case has the stereotype «variant». A variant use case is a use case which functionality can vary between elements of the product line. Figure 1 shows the use case for the messaging domain of the GoPhone product line. From the model it is possible to observe that *send message* is a variant use case of the product line. Further details regarding the use case variability are specified textually, in the use case description. Figure 2 is an extract from the textual documentation of the *send message* use case in the GoPhone report [7].

The *send message* description shows all the variation points of the use case. Variation points are identified by *OPT* or *ALT* tags. This approach explicitly points out all variation points of the use case but has disadvantages. For instance, if the use case is long, it may become very difficult to recognize a possible scenario for a member of the product line. Even further, this textual description is not adequate when the aim is the automation of tasks or the adoption of tools for dealing with variability.

In order to ease the automation of transforming the requirements of a product line into its high-level architecture (i.e., apply the 4SRS technique) we propose the explicit representation of the variation points in the use case model. In order to do so, a careful analysis of the initial use cases must be done.

The initial use cases, that are used to communicate the system functionalities with the stakeholders, must be transformed in order to express explicitly the functional variations of the product line. This activity can be done without the intervention of the users of the system. The main idea is to extract *include* and *extend* relationships from the textual description of the use cases. The *include* relationships will result from functional decomposition and will allow the discovery of functional commonalities among use cases. The *extend* relationships will basically result from extracting alternative and optional functionality from the use cases.

We like to view these activities as the construction of a three dimensional space representing the functionality of the product line: commonality, detail and variability. For instance, for each use case, we can go deeper (y axis) and broader (x axis) by adding detail as we do functional decomposition and find commonality. In a third dimension (z axis) we can express variability. This approach simplifies use case diagrams when requirements are extensive and complex because, for a given use case, one can choose to view only one perspective from the three dimensional space. In our approach we focus only on product line variability, i.e., functionality that can vary according to product line members. Variability that is common to all members of the product line can also be represented in the use case diagrams. But this can clutter the diagrams. We also advocate that this kind of variability can be better expressed in other types of diagrams like, for instance, activity diagrams. In this paper we will only address product line variability.

Next we briefly present how to construct the three dimensional space of use cases.

Functional decomposition
The initial use cases of the product line should be developed following, for instance, the process described by Alistair Cockburn [9]. This should result in use cases with a main scenario description similar to the one presented in Figure 2. These use cases should be at a medium level of detail, also know as *user level*. Based on these initial use cases, an analysis should be made with the goal of factor out fragments that have high degrees of commonality between them. For instance, regarding the messaging domain of the GoPhone product line we have found three of such fragments that have become the use cases `{U0.1.1} Choose Recipient` (steps 10 and 11 of Figure 2), `{U0.1.2} Compose Message` (steps 3 to 8 of Figure 2) and `{U0.1.3} Send Message to Network` (steps 12 to 14 of Figure 2). These use cases are common to the initial use cases `{U0.1} Send Message` and `{U0.2} Start Chat`. According to the 4SRS technique, each use case name is prefixed, within curly brackets, with a 'U' followed by period separated numbers denoting the level of the use case.

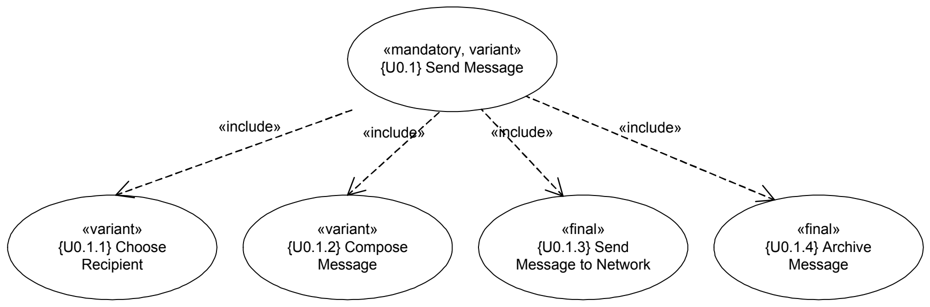


Fig. 3. Decomposition of use case `{U0.1} Send Message`

We adopt Gomaa's notation [1] for classifying use cases regarding their inclusion in the product line. As such, use cases can be marked with the stereotypes *mandatory*, *optional* or *alternative*. A mandatory use case is a use case that has to be included in all members of the product line. Optional and alternative use cases are only included in the members of the product line according to an inclusion condition. Alternative use cases must be in a group where usually one of the use cases is the default. This classification provides a very good foundation for viewing and analyzing the use case model according to the features of possible members of the product line.

When decomposing use cases, it is best to express the conditions regarding product line membership in the relationships, not the use cases. The reason is that these use cases can be included in several parent use cases, and the inclusion can vary depending on the parent. In Figure 3, `{U0.1} Send Message` has the stereotype *mandatory*, stating that this user level use case is to be included in all members of the product line. All the included relationships are mandatory, meaning that the use case `{U0.1} Send Message` requires all of the included use cases. Regarding decomposability, the *final* stereotype indicates that the use case is not decomposable any further. We also propose the stereotype *abstract*, to mark use cases which have all their functionality realized by others use cases, as a result of the decomposition. Since the default stereotype for the include relationship is *mandatory*, the diagram of Figure 3 does not show this keyword near the relationships. To be noted that non-mandatory functionality regarding `{U0.1} Send Message` should be left to the variability perspective.

Variability externalization

The presented stereotypes do not provide hints regarding the variability of the use cases. So, in order to also express this information in the use case model we use the *variant* stereotype. When this stereotype appears on a use case it means that the use case has variability at the level of the product line. For instance, in Figure 3, use case `{U0.1.2} Compose Message` has the stereotype variant. This means that, at the product line level, this use case is variable. According to our three dimensional approach, Figure 4 presents `{U0.1.2} Compose Message` in the variability perspective (z-axis). The extension points of the use case are visible and also are the conditions of inclusion of the extending use cases, according to the UML 2.0 notation. The information required to construct these perspectives can be easily extracted from use case textual descriptions. For instance, all the information required for Figure 4 can be extracted from Figure 2.

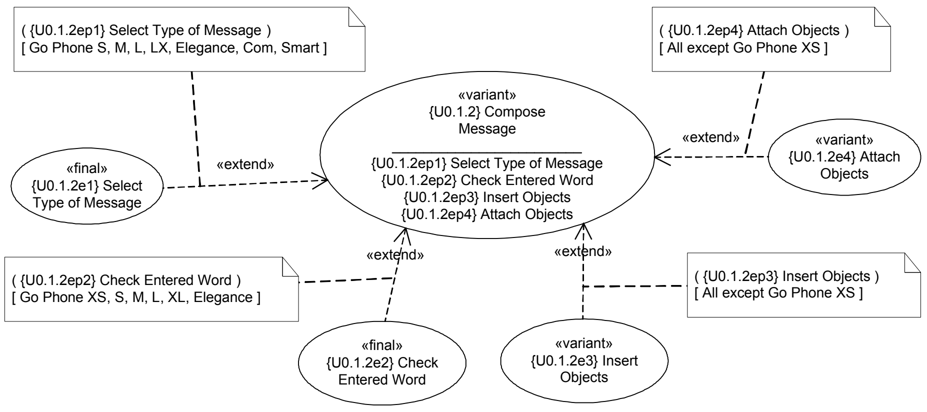

({U0.1.2ep1} Select Type of Message)
[Go Phone S, M, L, LX, Elegance, Com, Smart]
({U0.1.2ep4} Attach Objects)
[All except Go Phone XS]
«variant»
{U0.1.2} Compose Message
{U0.1.2ep1} Select Type of Message
{U0.1.2ep2} Check Entered Word
{U0.1.2ep3} Insert Objects
{U0.1.2ep4} Attach Objects
«extend»
«extend»
«variant»
{U0.1.2e4} Attach Objects
«final»
{U0.1.2e1} Select Type of Message
«extend»
«extend»
({U0.1.2ep2} Check Entered Word)
[Go Phone XS, S, M, L, XL, Elegance]
({U0.1.2ep3} Insert Objects)
[All except Go Phone XS]
«final»
{U0.1.2e2} Check Entered Word
«variant»
{U0.1.2e3} Insert Objects

Fig. 4. Variability perspective of use case `{U0.1.2} Compose Message`

3. Architecture Derivation

This Section presents the application of the 4SRS technique to the GoPhone product line use case models. We basically present a description of the transformational steps with some examples to better explain the involved transformations.

3.1 Step 1 – Object Creation

In this step, each use case originates three objects. This operation follows the same approach as RSEB, which proposes the creation of three objects for each use case: an *interface* object, a *control* object and a *data* object. For instance, in the example of Figure 3, the use case `{U0.1} Send Message` originates three objects: {`O0.1.i}`, `{O0.1.c}` and `{O0.1.d}`. This is an automatic step, and also a blind one, since each and every non-abstract use case originates three objects. Each object is named according to the corresponding use case with a suffix that identifies the type of object.

Regarding the original technique, the adaptation required for dealing with product lines is the need to detail the use case diagrams with all the extension points. For instance, in the GoPhone case, this detail is never exposed in the use case model. The variability points are only described within the use case main scenario.

3.2 Step 2 – Object Elimination

This step of 4SRS is aimed at eliminating the unnecessary objects that resulted from the previous step. After this step, the object model should have only the objects that are functionally required, according to the requirements of the product line. The original 4SRS technique also states that "this step also supports the elimination of redundancy in the user requirements elicitation, as well as the discovering of missing requirements".

This is a major step of 4SRS and is comprised of several micro-steps.

Micro-step 2i: use case classification
In this micro-step, each use case is classified according to the *interface-control-data* heuristic that was used to automatically generate the objects in the previous step. The idea is that the classification of a use case can be a hint to eliminate unnecessary objects. Use cases are then classified according to one of the possibilities: "Ø", "i", "c", "d", "i-c", "i-d", "c-d", "i-c-d". Each letter is associated with one of the *interface-control-data* possibilities: "i"-*interface*, "c"-*control* and "d"-*data*. For instance, `{U0.1.4} Archive Message` is classified as "d", while `{U0.1.2e2} Check Entered Word` is classified as being "c-d".

Micro-step 2ii: local elimination

This micro-step regards the possible elimination of objects following the classification of the use cases in the previous step. To assist in this task, the description of the use cases should be used. For instance, the use case `{U0.1.2e2} Check Entered Word`, that was classified as being of type "c-d", is described in the GoPhone report as "If T9 is activated, the system compares the entered word with the dictionary". The value of this use case is based on the T9 functionality for validating and suggesting words. As such, the control and data facets are much more important than the interface. According to this, the object `{O0.1.2e2.i}` is removed from the object model.

Micro-step 2iii: object naming

This micro-step aim is to give proper names to objects that were not removed in the previous micro-step. Names can be derived from the base use case name, the description of the use case and also the classification of the object. For instance, object `{O0.1.2e2.d}` is named as `Word Repository`.

Micro-step 2iv: object description

All the existing objects should have a description. According to 4SRS, this description should be based on the use case description from which they resulted. Next we present an example of such a description.

{O0.1.2e2.c} Word Validator: *This object checks words as they are entered by the user. This functionality is typical of phones that have the "T9" feature. For checking and memorization of words, the object uses object {O0.1.2e2.d} Word Repository.*

Micro-step 2v: object representation

The aim of this micro-step is to globally validate the model. For instance, redundancy can be discovered and removed. Basically, this step performs a semantic validation of the object model and also of the use case model. For instance, objects `{O0.1.2e3e2.d} Picture Insertion`, `{O0.1.2e3e1.d} Draft Text Insertion`, `{O0.1.2e4e2.d} File Attach` and `{O0.3e3e1.d} File View and Save` all represent the functionality of a repository of files. As such, we maintain only `{O0.1.2e4e2.d} File Attach`, since the semantic of this object includes the functionality of the other three objects.

Micro-step 2vi: global elimination

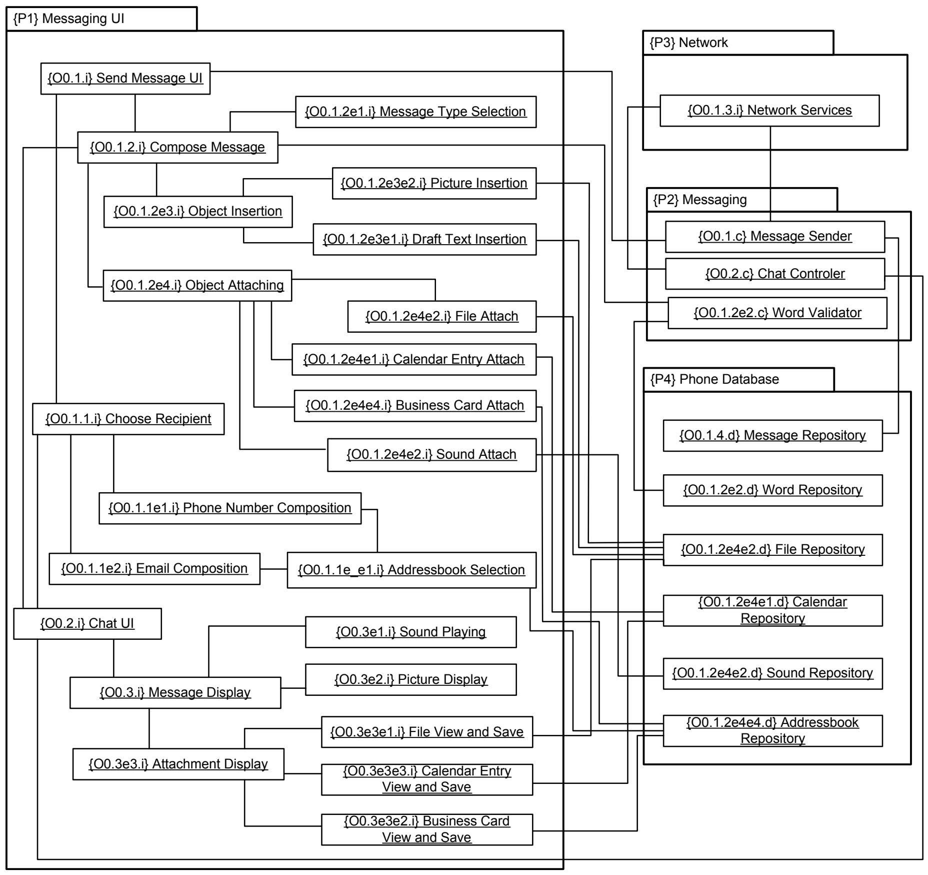
This is an "automatic" micro-step, since it is based on the results of the previous one. This step eliminates all the objects that were considered redundant in the previous step. For instance, resulting from the last micro-step, the objects `{O0.1.2e3e2.d}`, `{O0.1.2e3e1.d}` and `{O0.3e3e1.d}` are removed, since its functionality can be provided by the object `{O0.1.2e4e2.d} File Attach.` The result of this micro-step is a minimum number of objects that represent the product line functional requirements.

Micro-step 2vii: object renaming

The aim of this micro-step is to rename the objects that were not removed in the previous micro-step and that represent other objects. The documentation of such objects must also be updated. For instance, the `{O0.1.2e4e2.d} File Attach` object is renamed `{O0.1.2e4e2.d} File Repository` to proper represent its functionality, taking into account all the previous objects it represents.

3.3 Step 3 – Object Packaging & Aggregation

In this step, objects that make sense to be treated in a unified way can be placed in the same package. Aggregation can also be applied if there is a strong relationship between objects. This is usually the case of legacy objects in a sub-system. In the GoPhone product line this is not the case.



{P1} Messaging UI
{O0.1.i} Send Message UI
{O0.1.2e1.i} Message Type Selection
{O0.1.2.i} Compose Message
{O0.1.2e3e2.i} Picture Insertion
{O0.1.2e3.i} Object Insertion
{O0.1.2e3e1.i} Draft Text Insertion
{O0.1.2e4.i} Object Attaching
{O0.1.2e4e2.i} File Attach
{O0.1.2e4e1.i} Calendar Entry Attach
{O0.1.2e4e4.i} Business Card Attach
{O0.1.1.i} Choose Recipient
{O0.1.2e4e2.i} Sound Attach
{O0.1.1e1.i} Phone Number Composition
{O0.1.1e2.i} Email Composition
{O0.1.1e_e1.i} Addressbook Selection
{O0.2.i} Chat UI
{O0.3e1.i} Sound Playing
{O0.3e2.i} Picture Display
{O0.3.i} Message Display
{O0.3e3e1.i} File View and Save
{O0.3e3.i} Attachment Display
{O0.3e3e3.i} Calendar Entry View and Save
{O0.3e3e2.i} Business Card View and Save
{P3} Network
{O0.1.3.i} Network Services
{P2} Messaging
{O0.1.c} Message Sender
{O0.2.c} Chat Controler
{O0.1.2e2.c} Word Validator
{P4} Phone Database
{O0.1.4.d} Message Repository
{O0.1.2e2.d} Word Repository
{O0.1.2e4e2.d} File Repository
{O0.1.2e4e1.d} Calendar Repository
{O0.1.2e4e2.d} Sound Repository
{O0.1.2e4e4.d} Addressbook Repository

Fig. 5. Object model of the messaging domain

Since we are dealing only with the messaging domain of the product line, the packaging of objects follows this fact. As such, objects representing the user interface of the messaging domain are packaged in `{P1} Messaging UI` and objects representing messaging controlling and behavior are packaged in `{P2} Messaging`. Objects which major functionality is data persistence are included in `{P4} Phone Database`. We call this package *phone database* and not *messaging database* because it archives data regarding not only messages but other phone concepts like, contacts or files. `{P3} Network` is a package that includes objects with functionality regarding the mobile network, i.e., they represent the interface between the mobile phone and the network.

3.4 Step 4 – Object Association

This step introduces associations between objects that can be obtained from micro-step 2i. Also the relations between use cases can be used to generate associations between objects.

This is the last step in the 4SRS technique. Figure 5 presents the resulting object model for the messaging domain, including the packages. This object model, which resulted from the application of the 4SRS technique, is a system level object model. It provides high-level guidelines for the next phases of the development process. As such, it provides the basis for the requirements of a logical architecture that will support the following development phases. As it is possible to observe in Figure 5, the object model that result from the 4SRS technique includes all the functionality described in the source use cases. It is even possible to expose some hints regarding the product line variability, because, for instance, objects with an 'e' in their identification resulted from extending use cases. In the next Section we explore some issues regarding the logical architecture of a product line, namely variability representation and product member instantiation.

4. Logical Architecture

The major aim of a logical architecture is to serve as the basis for the design of a system. As such, it encompasses the description of the logical components of the system and also the interactions between them [10]. As presented in the previous Sections, the object model that results from 4SRS contains the components (objects) and interactions between them (object associations). As such, the object model that results from the 4SRS technique can be of great value for a system architect, because it clearly provides 'suggestions' for the logical components of a system and the interactions between them. This is very different from the usual gap that exists between requirements and the initial architecture for a system. This gap can be very 'dangerous' when the problem domain is new and there is not much knowledge in the solution space of the domain. In these cases it can be very difficult to design the system or even apply design patterns.

In the GoPhone technical report, the product line architectural design is based on the KobrA method and also on two patterns: the *mediator pattern* and the *state pattern*. The objective of the mediator pattern is to achieve changeability and extensibility of the components and, as such, achieve flexibility in the product line. The justification for the state pattern is that it enables handling the small displays of mobile phones. These two patterns result from non-functional requirements: flexibility and state management. They impose some guidelines in the architecture but they do not provide information regarding the functional components of the architecture. This is what we propose to achieve with the adoption of the 4SRS technique: a semi-automatic technique to obtain the product line's architecture functional requirements. The object model presented in Figure 5, which resulted from applying the 4SRS technique, depicts a partial view of such requirements for the GoPhone system. With such a model it is possible to design the system by applying well-known patterns, such as the mediator and the state pattern (such as in the GoPhone report). The difference from the GoPhone report is that, with our approach, we know which logical functional components are necessary to incorporate in the design. In this case, our logical architecture for the GoPhone product line is very similar with the one from the original report, the major difference being the fact that in our process we did not adopt KobrA.

The 4SRS technique was originally designed for obtaining the logical architecture of single systems. For this reason it does not deal explicitly with variability. As we saw, the main resulting artifact of the 4SRS technique is the object model. In our experimental approach to adapt 4SRS for product lines we have already proposed the need to externalize variability in the use case model. Regarding the logical architecture we also propose that other views of the system are needed to properly address product line development requirements. For instance, a class model may be more appropriate to express variability at the architectural level. Also, activity models are more appropriate to express fine grained variability. As such, we propose a multiple model approach for 4SRS. A similar approach can be also find in [11].

This multiple model approach is also more suited to deal with product line member instantiation. Product line member instantiation is based on the selection of features required for the member being instantiated. As mentioned in Section 1, the usual approach is to build a feature diagram to guide this instantiation. The construction of a feature diagram can be done in parallel with the use case diagrams. In our approach, feature diagrams correspond to choices in the variability perspective (z-axis), when navigating through the use case model. A functional feature is basically realized by a use case. Extending use cases become optional or alternative features. Figure 6 presents a feature diagram for *send message*. The Figure also presents a possible example of the selection of features for a product line member, by showing them in gray.

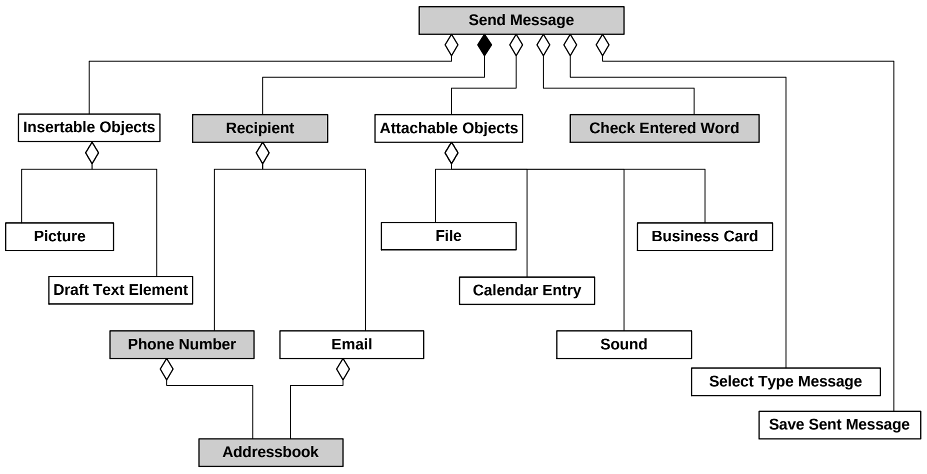


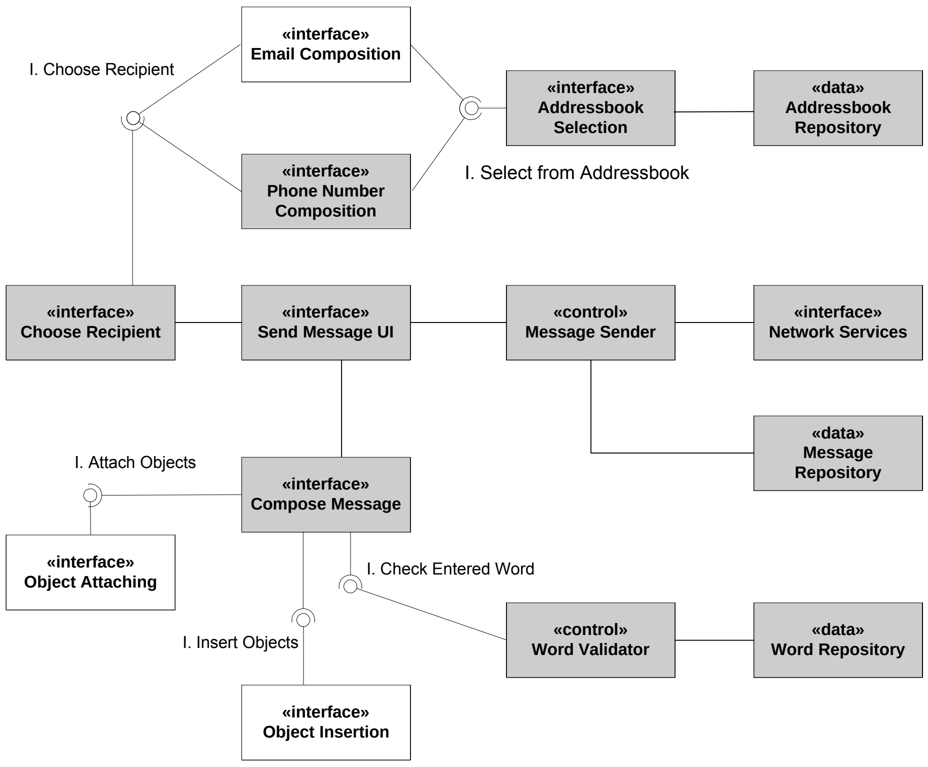
Fig. 6. Feature diagram for *Send Message* (Based on the notation proposed in [1])

Figure 7 presents an excerpt of a possible class diagram depicting the *send message* feature according to the feature selections of Figure 6. The class model should be constructed after the object model. The major goal of the class model, at this logical architectural level, is to be the first approximation to a meta-level structural model of the product line architecture. In the process of constructing the class model it is possible and even common that functionalities provided by several objects become realized by a single class or a hierarchy of classes.

Similar to the object diagram, the class diagram at this logical architectural level is used to represent the product line at a component level of abstraction. For the moment, we are not adopting component diagrams because they are best suited for modeling the system at a lower level of abstraction, particularly at the physical level.

The class model also provides a way to explicitly represent the product line variability. So, the construction of the class model is also based on the use case model and feature model. The description of the process for the construction of the class model is out of the scope of this paper. We intend to address it in our future work.

As it is possible to observe in Figure 7, in this experimental approach we have adopted outgoing and incoming interfaces to model extension points. This seams to be an appropriate choice at this logical component level. This option does not compromise later design decisions of how to realize the extension points. In fact, other authors have proposed comprehensive feature variability realization techniques at the design level that are based on interfaces [12].



«interface» Email Composition
I. Choose Recipient
«interface» Addressbook Selection
«data» Addressbook Repository
«interface» Phone Number Composition
I. Select from Addressbook
«interface» Choose Recipient
«interface» Send Message UI
«control» Message Sender
«interface» Network Services
«data» Message Repository
I. Attach Objects
«interface» Compose Message
«interface» Object Attaching
I. Check Entered Word
«control» Word Validator
«data» Word Repository
I. Insert Objects
«interface» Object Insertion

Fig. 7. Excerpt of class diagram for *Send Message*

Since in our approach there is a very direct mapping between the use case model and the feature model and because it is easy to keep trace links from the class model to the object model and ultimately to the use case model, it is possible to derive the architectural requirements for a product line member based on its features.

5. Conclusions

In this paper we have explored an approach for deriving a software product line logical architecture from its requirements, by adopting and adapting a transformational technique. We have focused the discussion in the transformational technique for obtaining an object model from the use case model. This is only a part of a multi-view and multi-model process approach for product line development. We intend to present and discuss more aspects of this process in our future work.

We have found the results of this experimental approach very promising. Nonetheless, several questions still remain open and require further validation. For instance, the tree dimensional view of the use case model seems to be a requirement for dealing with very complex product lines. But we need to validate this with more case experiences. This will also provide a context to further explore the technique for feature diagram construction based on the use case variability perspective.

Another point open to further research regards variability representation. For the moment our approach only deals with component level variability. It seams, even in the GoPhone case, that more fine grained representation for variability is needed. This is true for use cases, object and class diagrams. In our future work we intend to approach this problem mainly by using activity diagrams in the context of use cases and also explore aspect oriented approaches as a way to deal with operation level variability. OCL seams also a promising approach to express variability in the model in a more formal way.

References

[1] Gomaa, H., *Designing Software Product Lines with UML*. Addison Wesley. 2005.

[2] Kang, K.C., J. Lee, and P. Donohoe, *Feature-Oriented Product Line Engineering.* IEEE Software, (July/August 2002). 2002.

[3] Griss, M.L., J. Favaro, and M. d'Alessandro. *Integrating Feature Modeling with the RSEB*. in *Fifth International Conference on Software Reuse*. Victoria, Canada. IEEE Computer Society Press. 1998.

[4] Jacobson, I., M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley Longman. 1997.

[5] Anastasopoulos, M., J. Bayer, O. Flege, and C. Gacek. *A Process for Product Line Architecture Creation and Evaluation*. IESE. Technical report: 038.00/E. 2000.

[6] Machado, R.J., J.M. Fernandes, P. Monteiro, and H. Rodrigues. *On the Transformation of UML Models for Service-Oriented Software*. in *ECBS International Conference and Workshop on the Engineering of Computer Based Systems*. Greenbelt, Maryland. 2005.

[7] Muthig, D., I. John, M. Anastasopoulos, T. Forster, J. Dorr, and K. Schmid. *GoPhone - A Software Product Line in the Mobile Phone Domain*. IESE. Technical report: 025.04/E. 2004.

[8] Bayer, J., D. Muthig, and B. Gopfert. *The Library Systems Product Line - A KobrA Case Study*. IESE. Technical report: 024.01/E. 2001.

[9] Cockburn, A., *Writing Effective Use Cases*. Addison-Wesley. 2001.

[10] Garlan, D. and M. Shaw. *An Introduction to Software Architecture*. Carnegie Mellon University. Technical report: CMU-CS-94-166. 1994.

[11] Gomaa, H. and M.E. Shin. *A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines*. in *ICSR International Conference on Software Reuse*. Madrid. 2004.

[12] Lee, K. and K.C. Kang. *Feature Dependency Analysis for Product Line Component Design*. in *ICSR 2004 - International Conference on Software Reuse*. Madrid. 2004.

Supporting the Modeling of Embedded Systems

Vesna Milijic
Department for Applied Computer Sciences
Catholic University Eichstätt-Ingolstadt, 85072 Eichstätt, Germany
vesna.milijic@ku-eichstaett.de

Abstract

A new approach for modeling embedded systems is presented that uses a grammar to derive a specification of an embedded system. The specification is transferred to an intermediate model that corresponds to a Signal Petri net model with specific properties. These properties assure that the resulting Signal net model depicts an embedded system. The approach is exemplarily applied to model a garage door mechanism.

1 Introduction

For developing high-quality control systems including a high-quality documentation, formal methods are necessary to perform system analysis, verification and simulation. As various modeling techniques and formalisms exist in the area of system engineering, the development of systems in industrial practice is thought to be supported well. However, we still face incorrect systems and problems in system engineering.
In parts, this is due to not using the existing formalisms: an industrial case study (see [2]) showed that creating and working on models is not an easy task. Finally, designers often keep the usual development process of writing down a specification document and implementing an ad hoc control program. So the main reference of a system is not a formal system model, but an informal specification document. This document suffers from being incomplete, incorrect and being not up-to-date.
In contrast, if modeling is part of the system engineering process, other problems arise.

- Even if modeling is done in industrial practice, the first step is to write a specification document of the system in natural language. After that, the modeler or someone else will translate this document into a system model. In this step some of the statements of the document turn out not to be suitable for specifying the system because they deliver contradictory, false or insufficient information. So the modeler is forced to iterate the time-consuming procedure of first specifying and then modeling the system until it seems probably well modeled. As there is no direct transfer between the specification document and the model, the modeler can only guess if the model meets the specification; they are not associated with each other automatically.

- Modeling formalisms are written for universal applicability. The modeler usually designs only a certain *type* of systems (e.g. embedded systems), but by the means of the modeling formalism he risks to create models that do not belong to the desired type (e.g. workflow systems). Thus, while trying to create a system out of the ideas listed in the specification document, the modeler has to spend much attention to miss out the types of systems he is not interested in.

- Some formalisms model at once the structure of a system and its behaviour (e.g. Petri nets and their derivatives). They are especially interesting as it is sufficient to have one single model for every kind of analysis. However, modeling is more difficult, as any alteration of the model can affect both structure and behaviour.

To overcome these problems, a new approach of system engineering is proposed.

1.1 The General Approach

Despite the problems concerning the specification document, it is very important in the practice of system engineering. Using natural language, it leads to a basic understanding of the system's functionalities. Thus, it should not be disposed - by the way, in the area of requirements engineering you can find new approaches that tend to express formal requirements in natural language (see [9]).
However the "formalism" of specification documents, namely natural language, must be restricted: Only components and combination of these that describe the desired system type will be accepted. For this reason, the concept of sublanguages propagated by computer linguistics (see [6]) will be adopted: By offering construction rules (in our case given by a context-free grammar including global variables), a limited portion of all possible natural language statements and texts are derived: Descriptions that are created out of that grammar are specifications of the desired system type (left side of figure 1).
As a direct translation into the chosen modeling formalism is not possible, an intermediate formalism is needed. By defining chunks of the atomic elements of the original modeling formalism, we create semantic units that are

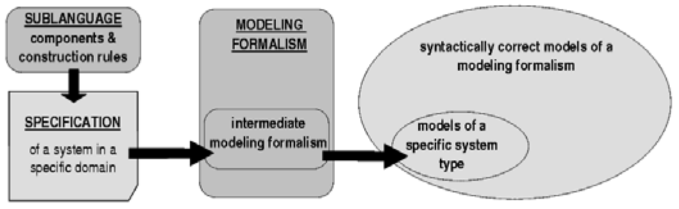

SUBLANGUAGE
components & construction rules
SPECIFICATION
of a system in a specific domain
MODELING FORMALISM
intermediate modeling formalism
syntactically correct models of a modeling formalism
models of a specific system type

Figure 1: Design of embedded systems: a new approach

the basic elements of the intermediate formalism (denoted by the middle part of figure 1). All components and combinations of the grammar must be modeled in the intermediate formalism.
Finally, the formal model of the desired system type is obtained (right side of figure 1).

1.2 In the Area of Embedded Systems

This approach is highly suitable for the area of embedded systems where the need of an adequate formal model arises from the systems' behavioural complexity. As modeling formalism extended Signal nets are used (a derivation of Petri nets).
The next chapter will provide a definition of embedded systems and illustrate why extended Signal nets are an adequate modeling formalism. After that, chapter 3 will define a sublanguage for specifying the behaviour of embedded systems. Chapter 4 is concerned with the intermediate formalism and in chapter 5, an example modeling case is showed. Finally, chapter 6 summarizes and gives a short forecast of future work to be done.

2 Embedded Systems and Signal Nets

2.1 Defining Embedded Systems

According to [7], a *reactive system*, which is superordinate to embedded systems, is an information system that continually interacts with its environment. Unlike transformational systems that calculate a result from input data and then reach a terminal state, most reactive systems are intended to run infinitely. Usually they are non-sequential distributed systems for monitoring and control tasks. They contain several elements (software and hardware) that communicate by signal exchange. Reactive systems can be differentiated in circuits, process computers, and embedded systems ([1]). Between those, the importance of interaction from environment to system varies: While circuits are characterized by minor interaction with their environment, process computers (e.g. PLC) work with parameters and respond

to environmental influence. Interaction with the environment is most important for embedded systems. As figure 2 shows, these systems are usually integrated in a physical environment (embedding system) and, except the user interface, they are hidden from the user. Notice that the behaviour of an embedded system is completely defined by the internal state of the system and the environmental influence, e.g. the manipulation by the user or by the embedding system. However, since any combination of internal state and environmental influence can cause system behaviour, embedded systems possibly show strong behavioural complexity.
Examples of embedded systems are technical devices (like washing machines, coffee machines, antilocking systems) or building automation (heating or ventilation).

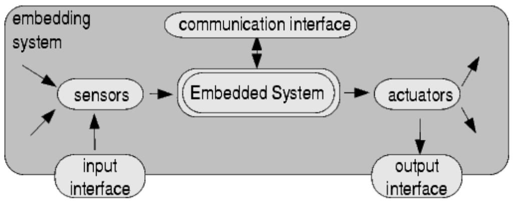


Figure 2: Schema of Embedded Systems

2.2 The Choice of Signal Nets

Extended Signal nets (an extension of Petri nets) were chosen as modeling language because they fit well regarding the properties of embedded systems:

- The use of Petri nets is generally a good solution for modeling distributed systems. In contrast, state based approaches are not sufficient for larger embedded systems because of the possibly huge state space. Furthermore, the property of being executable will enable model validation by simulating the system behaviour.
- Extended Signal nets are an adequate modeling formalism for embedded systems as they support the concept of communicating and initializing system behaviour by signal exchange with signal arcs. Other types of arcs perform reading, writing or checking tasks. The use of high-level places enables modeling of sensor data in embedded systems. Finally, extended Signal nets distinguish between external places and transitions and internal ones, what corresponds to the differentiation of plant and control in the case of embedded systems.

Assuming that the formalism of Petri nets is generally well-known, only the specific features of extended Signal nets that are interesting for modeling

embedded systems will be explained next. For a more detailed description see [3, 8].
A Signal arc (see figure 3) connects two transitions. If an enabled transition is connected to another enabled transition by a signal arc, both transitions fire synchronously (the first transition synchronizes the second transition). A transition with an incoming signal arc will never fire without the synchronization signal, whereas a transition with an outgoing signal arc can also fire alone. If two or more signal arcs lead to one transition, this transition will fire at the first arrival of a signal (OR-semantics).
Signal nets offer a possibility to check relevant markings. This is done by test arcs (figure 3): A transition, connected by a *read arc* to a place (no high-level place), can only fire when the place is marked. A transition connected to a place by an *inhibitor arc* fires if the place is *not* marked. Test arcs never cause a marking change of the respective place.
Two types of places (figure 3) are allowed in the formalism of Signal nets:

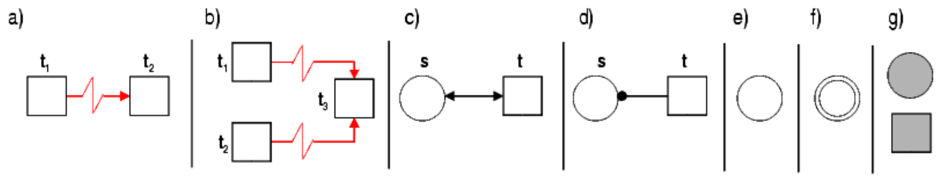


Figure 3: a) Firing of the transition t_1 triggers firing of transition t_2 b) Transition t_3 will fire at the first firing time of either t_1 or t_2 c) read arc d) inhibitor arc e) low-level place f) high-level place g) external state and transition

Low-level places that are depicted by a simple circle and contain only a simple token type (a black token), and high-level places. The latter are drawn as double-framed circles. For the token domain, we restrict to real numbers.
The gray color of places or transitions indicates that they are external.

3 A Grammar for Specifying Embedded Systems

What is an adequate specification language for embedded systems? According to Harel, it must respect the modelers' kind of thoughts (see [5]): Specification sheets with statements of natural language that depict behavioural characteristics of the system. For example, *If you push the button 'down' of the remote control, the garage door will close.* is such a behavioural specification. Thus, the statements of our specification language should describe behavioural characteristics. Additionally, since we restrict to model components and combinations of components of embedded systems, only the

behaviour of those should enter the specifications (see figure 1). However, since a generally accepted specification of embedded systems does not exist, the invented specification language can only be accounted as a simple proposal that has to be further evaluated.
Typically, the systems to be modeled are composed of two parts: A model of the plant that corresponds to the physical part of the embedded system and a model of the control. The physical part is necessary to model the environmental influence on the system' s behaviour. It consists of the set of possible control elements as actors (for example a switch to switch on or off the coffee machine, a rotary switch to set the washing program to 30 or 60°C) and of sensors that are needed to invoke a control algorithm (for example 'if the heat sensor' s value is over 23°C, air conditioning has to start').
Notice that a description will be obtained that specifies a valid embedded system but this might not be the desired one. Also, the problem of incomplete or inconsistent specification documents is not solvable by the use of the grammar. Surely, the sublanguage could be strongly limited to avoid undesirable behaviour, but this would diminish the set of derivable specifications. Since this is not intended, the resulting models must be validated (e.g. by construction of system runs).
The rest of the chapter is devoted to the description of the possible statements. Tables 1 to 12 and figure 5 show the grammar and the global variables. Nonterminal symbols are written in typewriter font and capital letters while terminal symbols are written in small letters. Global variables are typed italic. The symbol '|' is separating two different production rules of one nonterminal symbol.

3.1 Specifying Control Elements

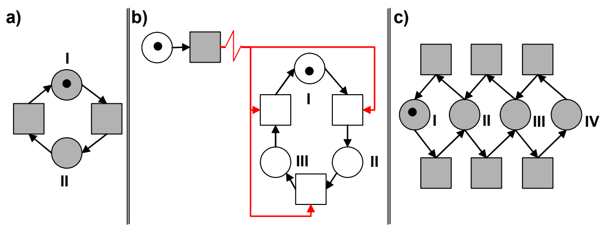


Figure 4: a) switch with two states b) pushbutton: no visual changing, but internally the state changes into one direction c) rotary switch with starting and ending point

First, relevant control elements have to be modeled. By the rules of table 1, for each element an adequate representation is chosen (figure 4

depicts some control elements) and all possible states or values have to be defined. By 'TYPE', the controllability of the element is defined. If it is controllable only by external manipulation, the control algorithm is not allowed to change the state of the element.
When all control elements are defined, the rule 'S ⇒ BEHAVIOUR' is called which denotes the starting point of the behavioural description.

S	⇒ELEM is TYPE. S | BEHAVIOUR
ELEM	⇒switch/ button *id, states, default state* | pushbutton *id, states, default state* | rotary switch *id, states, default state* | key panel *id, states, default state* | display *id, values, default value* | sensor *id, values, default value* |
TYPE	⇒only controllable by external manipulation| controllable by both external and internal manipulation

Table 1: Specifying Control Elements

3.2 Behaviour of the control algorithm

Every 'BEHAVIOUR' denotes a behavioural characteristic of the embedded system. By the first rule of table 2, further 'BEHAVIOUR's can be added. In the middle part, the rules define system behaviour of the control algorithm that is independent from control elements. The lower part introduces behavioural scenarios that deal with dependencies between different parts of the system. They will be subject of 3.3.

BEHAVIOUR	⇒BEHAVIOUR BEHAVIOUR | UNIT occurs. SEQUENCE| Every x seconds UNIT occurs. | x times UNIT occurs. SEQUENCE| Repeatedly UNIT occurs. | UNIT and UNIT occur concurrently. SEQUENCE| If SYSTEM-STATE2 CONDITION1, then CONSEQUENCE occurs. SEQUENCE| If CONDITION2, then CONSEQUENCE occurs. ELSE SEQUENCE| If SYSTEM-STATE1 CONDITION1, then CONSEQUENCE occurs. ELSE SEQUENCE

Table 2: Specifying Behavioural Characteristics

Every statement that specifies behaviour of solely the control algorithm contains the nonterminal symbol 'UNIT'. A 'UNIT' can denote an arbitrar-

ily big part of the control algorithm. It can be substituted by a simple 'ACTIVITY' (i.e. a firing sequence or a measurement of values, see figure 5) or it defines some system behaviour on a lower level when it is substituted by '*execution of unit ...*'. So a top-down structuring of the control algorithm is possible by recursively modeling new 'UNITs' that are part of existing 'UNITs'. A structuring within one level can be reached by modeling more than one 'BEHAVIOUR' and replacing them by different 'UNITs'.
As table 3 shows, 'UNIT' can also be used to reset another 'UNIT' or a con-

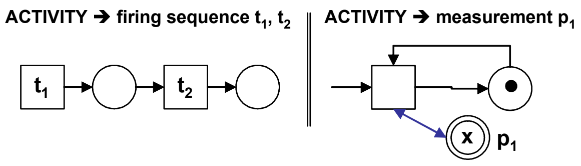


Figure 5: Exemplary substitutions of ACTIVITY

trol element in its initial state. The construction rule for a reset-'UNIT' is as follows: For each place p of a unit or element to be reset, construct a new transition t_p with an outgoing flow arc from the place to the corresponding transition. Additionally, construct an ingoing flow arc from all places that are initially marked to the corresponding transition. For starting the reset, a transition t_{start} is needed that triggers all other transitions t_p by signal arc connections (see figure 6 for an example).

UNIT	⇒ACTIVITY |
	execution of unit *id*, description: BEHAVIOUR |
	reset of unit *id* | reset of element *id*

Table 3: Substitution Rules for UNIT

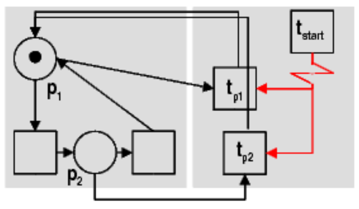


Figure 6: On the left side a UNIT is depicted. The right side shows the corresponding reset net

As denoted in the middle part of table 2, a 'UNIT' can occur once or repeatedly. By some statements, an endless occurrence is indicated (e.g.

'*Every x seconds* `UNIT` *occurs*'), other statements describe an ending behaviour: In such a case, '`SEQUENCE`' is appended to enable the modeling of arbitrary many sequences by the rules given in table 4.

`SEQUENCE`	$\Rightarrow \lambda$ | `SEQUENCE` `SEQUENCE` |
	After that `CONSEQUENCE` occurs. |

Table 4: Substitution of `SEQUENCE`

3.3 Causal Related Behaviour

For modeling causal related behaviour, the lower part of table 2 is relevant. These rules represent the initial point of all behavioural characteristics dealing with dependencies and can be divided into a condition and a consequence section and - if needed - a section that models events that occur in case of violated conditions. Let us have a look at each section separately.

3.3.1 Condition Section

The condition section always starts with *If...* and depicts a situation which is characterized by two components: The first component is able to *initialize* the consequence section (see table 5) and the second describes a *part of the given system state* that provides permission to the first component for effectively doing initialization (see table 6).

`CONDITION1/2`	$\Rightarrow$ `TIME` `STATE` is given |
	f(`MP`) is in the range of *lb* to *ub* |
	f(`MP`) is in the range of *lb* to *ub* and `CONDITION1` |
	`UNIT` starts | `UNIT` ends *(both only CONDITION1!)* |
	`UNIT` starts while `STATE` is given |
	`UNIT` ends while `STATE` is given |
	`CONDITION1` or `CONDITION1` *(resp. COND2 or COND2)*

Table 5: Rules for `CONDITION1` and `CONDITION2`

Initializing Component The first substitution rule of table 5, '$\Rightarrow$ `TIME` `STATE` *is given*', characterizes a state-based initialization of behaviour. As '`STATE`' is substitutable according to table 7, finally a marking vector for a part of the net is specified.

By the substitution of '`TIME`' the modeler can decide if this condition has to exist for a certain amount of time (i.e. first rule of table 8) or not (i.e. by omitting '`TIME`' with the use of the λ-Rule).

`SYSTEM-STATE1`	$\Rightarrow$`SYSTEM-STATE1 SYSTEM-STATE1` | `SYSTEM-STATE1 SYSTEM-STATE2` | - within x seconds `Q` | - not until x seconds `Q`
`SYSTEM-STATE2`	$\Rightarrow\lambda$ | `SYSTEM-STATE2 SYSTEM-STATE2` | - when x seconds have passed `Q`
`Q`	$\Rightarrow$after `STATE` is given - | after `UNIT` has started - | after `UNIT` is finished -

Table 6: Substitution of `SYSTEM-STATE1`, `SYSTEM-STATE2` and `Q`

`STATE`	$\Rightarrow$(`STATE` and `STATE`) | (`STATE` xor `STATE`) | not `STATE` | *marking*

Table 7: Substitution of `STATE`

In the case of modeling some sensor data as initialization factor for a consequence section, the second and third substitution rule of table 5 are applied. By the input of a lower bound (*lb*) and an upper bound (*ub*), the modeler defines how to proceed depending on the measured value. As sometimes not only the value itself is relevant but a result of a mathematical function using one or more sensor values as parameters, 'f(`MP`)' denotes such a function using arbitrary many markings of places (according to the substitution of '`MP`' depicted in table 9).

The rules that start with '`UNIT` *starts/ ends ...*' denote event-driven starting points for future behaviour.
Of course, alternative conditions for initialization can be modeled by using the last rule of '`CONDITION1/CONDITION2`'.

A part of the given system state as initialization condition In specifying behavioural characteristics of embedded systems, the modeler sometimes needs access to former behaviour of the system. This access is realized by the statements that can be produced by the rules of table 6:

3.3.2 Consequence Section

The common starting point for the consequence section is '*..., then* `CONSEQUENCE` *occurs.*'. Table 10 shows the substitution rules for '`CONSEQUENCE`': It is possible to either model a single consequence, or, by the use of '`FURTHER`', a

`TIME`	$\Rightarrow\lambda$ | for x seconds

Table 8: Rules for `TIME`

`MP`	⇒`MP`, `MP` | *marking*

Table 9: Markings of Places

`CONSEQUENCE`	⇒`UNIT` `FURTHER` |
	after x seconds `UNIT` `FURTHER` |
	x times `UNIT` |
	every x seconds `UNIT` as long as `SYSTEM-STATE1` `CONDITION1` `FURTHER` |
	every x seconds `UNIT` as long as `CONDITION1` `FURTHER` |
	repeatedly `UNIT` as long as `SYSTEM-STATE1` `CONDITION1` `FURTHER` |
	repeatedly `UNIT` as long as `CONDITION1` `FURTHER`

Table 10: Substitution of `CONSEQUENCE`

set of consequences (according to table 11). Each consequence depicts a somehow specified occurrence of an '`UNIT`'. For all statements that imply possibly infinite behaviour, the modeler can introduce an ending condition '*... as long as*' plus a condition denoting this.

`FURTHER`	⇒λ | , `CONSEQUENCE`

Table 11: Rules for `FURTHER`

3.3.3 Violated Condition Section

The behaviour of the system when violating some condition is denoted by '`ELSE`'. If no special behaviour is desired in case of a not fulfilled condition, '`ELSE`' can be substituted by λ. If the modeler decides to define a behaviour in this case, he can use '`ELSE` ⇒ *Else* `CONSEQUENCE` *occurs.*'(table 12). Here again, '`CONSEQUENCE`' can be substituted by only one or a set of consequences as specified in the consequence section (3.3.2). The model for '*Else* `CONSEQUENCE` *occurs.*' heavily depends on the condition section: Let us first focus condition sections that contain only an initialization component but no additional system state as condition (i.e. statement '*If* `CONDITION2`*, then* `CONSEQUENCE` *occurs.* `ELSE` `SEQUENCE`' which does not include '`SYSTEM-STATE1`' or '`SYSTEM-STATE2`'): To continue system behaviour at the arrival of a condition, this condition must be evaluated to either *true* or *false*. If it is *true*, a consequence will occur, if it is *false*, an alternative behaviour (specified by the substitution of '`ELSE`' will be executed. However some condition sections cannot be evaluated to *false*. For example, this is the case for the condition '`UNIT` *starts*': At the occurrence of `UNIT` *starts*, the condition is evaluated to *true* but as long as `UNIT` did not start, it cannot be decided if the condition will be *true* in the future. Therefore,

ELSE	$\Rightarrow\lambda$ | Else CONSEQUENCE occurs.

Table 12: Substitution of ELSE

the condition section can never be evaluated to be *false*.
For this reason, 'CONDITION1' and 'CONDITION2' are differentiated: While 'CONDITION2' contains only conditions that can be evaluated to both *true* and *false*, 'CONDITION1' additionally contains conditions that cannot be evaluated to *false*.
Let us look at 'SYSTEM-STATE1' and 'SYSTEM-STATE2': The statements containing '*within x seconds after...*' allow an initialization of the consequence section if the condition is fulfilled within the given time. If this time has passed before the condition is fulfilled, the condition is violated. Contrary, by using '*not until x seconds ...*', the condition is fulfilled when 'CONDITION1/CONDITION2' happens after the given time has passed and violated when - at the occurrence of 'CONDITION1/CONDITION2' - it has not yet passed.
In contrast, the condition sections starting with '*when x seconds have passed...*' mean the following: They are *true*, if 'CONDITION1/CONDITION2' occurs any time after the given time has passed; everything that happens within this time is ignored. But such condition sections cannot be evaluated to *false*, so a differentiated use with respect to 'ELSE' is needed. Also in the case of '*If SYSTEM-STATE2 CONDITION1,...*', a substitution '*SYSTEM-STATE2* $\Rightarrow \lambda$' is admissible what possibly leads to a condition section that cannot be evaluated to *false*. Therefore, statements containing such conditions are subsumed to one nonterminal symbol ('SYSTEM-STATE2') which does not allow the use of 'ELSE', while 'SYSTEM-STATE1' contains all substitutions that can be evaluated to both *true* and *false*.
Note that arbitrary many substitutions of 'SYSTEM-STATE1/SYSTEM-STATE2' can be listed. Such a set of condition statements can possibly not be evaluated to *false* if it contains at least one statement of 'SYSTEM-STATE1' and the violated condition section is called the first time any of these has been evaluated to *false*. For allowing the consequence section to start by occurrence of 'CONDITION1/CONDITION2', every condition statement must be fulfilled.

4 The Intermediate Formalism

As previously written, the elements of the intermediate formalism are composed chunks of the basic modeling formalism. Furthermore, these components are composed in specific ways only. Figure 7 shows an example. The components and how to combine them has to be defined manually. After this, any specified net of the desired type can be modeled. By lack of space,

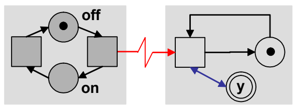


Figure 7: Intermediate model for: '*If the control element turns from off to on, then measurement of y occurs.*'

this paper will only show some example compositions in the modeling case.

5 A Modeling Example

For demonstrating the practical use of this approach, a control for a garage door is modeled in this section. By a remote control with two buttons (UP and DOWN), a motor can be controlled that ascends or descends the garage door. The motor is either running up, running down or in halt position. Two resistance sensors observe whether the door is completely open ('*RS_up*') and whether a resistance occurs on the downward direction ('*RS_down*'). The specification that is derived by the grammar is shown in the following two paragraphs:

Definition of the control elements. Switch/ button ("UP", states: "on"/"off", default: "off") is only controllable by external manipulation.
Sensor ("RS_up", values: 0 to 10, default: 0) is controllable by both external and internal manipulation.
Switch/ button ("DOWN", states: "on"/"off", default: "off") is only controllable by external manipulation.
Sensor ("RS_down", values: 0 to 10, default: 0) is controllable by both external and internal manipulation.
Switch/ button ("Motor", states: "run$\Uparrow$"/"stop"/"run$\Downarrow$", default: "stop") is controllable by both external and internal manipulation.

Behavioural characteristics. If not *Motor: stop* is given, then measurement of *RS_up* occurs. If not *Motor: stop* is given, then measurement of *RS_down* occurs.
If value of *RS_up* is in the range of 0 to 2 and *UP: on* is given, then *Motor: stop* $\rightarrow$ *run*$\Uparrow$ occurs. If value of *RS_up* is in the range of 3 to 10 and *UP: on* is given, then *Motor: run*$\Uparrow$ $\rightarrow$ *stop* occurs.
If value of *RS_down* is in the range of 0 to 2 and *DOWN: on* is given, then *Motor: stop* $\rightarrow$ *run*$\Downarrow$ occurs. If value of *RS_down* is in the range of 3 to 10 and *DOWN: on* is given, then *Motor: run*$\Downarrow$ $\rightarrow$ *stop* occurs.

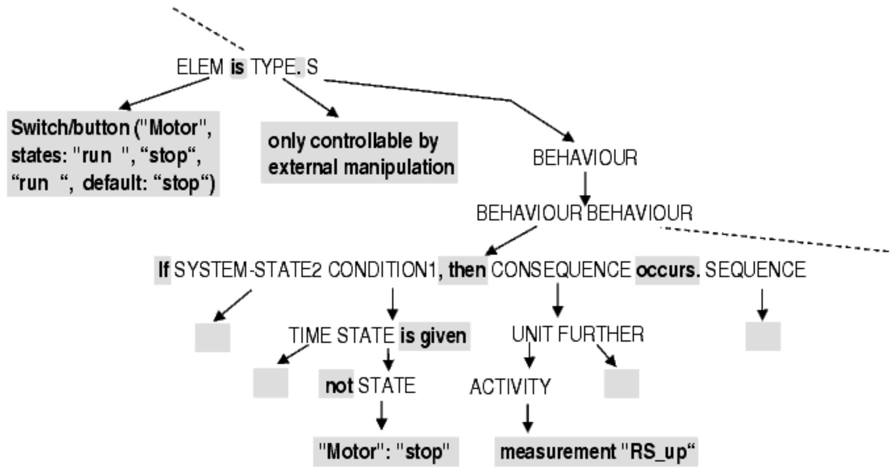

ELEM is TYPE. S
Switch/button ("Motor", states: "run ", "stop", "run ", default: "stop")
only controllable by external manipulation
BEHAVIOUR
BEHAVIOUR BEHAVIOUR
If SYSTEM-STATE2 CONDITION1, then CONSEQUENCE occurs. SEQUENCE
TIME STATE is given
UNIT FURTHER
not STATE
ACTIVITY
"Motor": "stop"
measurement "RS_up"

Figure 8: Generation tree for control elements

If *UP: on* $\rightarrow$ *off* starts or *DOWN: on* $\rightarrow$ *off* starts, then reset of *Motor* occurs.
Figure 8 depicts the generation tree for the control element "Motor" and the first behavioural characteristic. The gray boxes denote the final specification text. Figure 9 shows the resulting signal net model. The intermediate model is depicted by the lighter gray boxes and the connection between Signal net elements of different boxes.

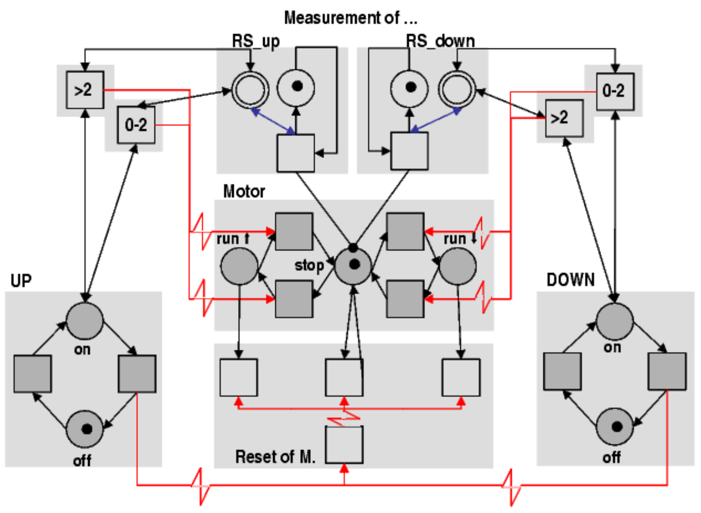


Figure 9: Signal net model of a garage door control

6 Conclusion and Future Work

This paper presented a method to support the modeling of embedded systems. A grammar including global variables was used to derive specifications that can be transferred into specific Signal nets (i.e. the intermediate model: extended Signal nets that represent embedded systems). Since this approach should be convenient for modelers in practice, tool support is needed to derive specifications out of the grammar. Also the invented grammar should be evaluated to point out its use in the description of embedded systems.
Furthermore, tool support is needed for defining new grammars for different system types and their translations into chunks of extended Signal nets.

References

[1] Broy, M., Spaniol, O.(Eds.) keyword: real-time system In *VDI-Lexikon Informatik und Kommunikationstechnik*, Heidelberg, Springer, 587 – 588, 1999.

[2] Desel J., Juhás G., Lorenz R., Milijic V., Neumair Ch., Schieber, R.: Modellierung von Steuerungssystemen mit Signal-Petrinetzen – eine Fallstudie aus der Automobilindustrie. In Schnieder, E. (Ed.): *8. Fachtagung Entwurf komplexer Automatisierungssysteme 2003*, Proceedings of EKA 2003, Braunschweig, Schwendowius, 273–297, 2003.

[3] Desel J., Milijic V., Neumair Ch.: Model Validation in Controller Design In *Lectures on Concurrency and Petri Nets, LNCS 3098*, Heidelberg, Springer, 467–495, 2004.

[4] Hanisch H.-M., Lüder A.: A Signal Extension for Petri nets and its Use in Controller Design. *Fundamenta Informaticae*, 41(4), 415–431, 2000.

[5] Harel, D., Marelly, R. *Come, Lets Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Heidelberg, Springer, 2003

[6] Sager J.C. *Language Engineering and Translation: Consequences of Automation*, Amsterdam, John Benjamins B.V, 1994

[7] Schneider, H.-J. (Ed.) keyword: system, reactive In *Lexikon der Informatik und Datenverarbeitung*, München, Oldenbourg, 850, 1998

[8] Starke P.H, Roch S. Analysing Signal-Net Systems. Informatik-Bericht 162, Humboldt Universität zu Berlin, Institut für Informatik, 2002

[9] Bitsch F. A Way for Applicable Formal Specification of Safety Requirements by Tool-Support. In *FORMS 2003 - Symposium on Formal Methods for Railway Operation and Control Systems*, 2003

A Modeling Language for Applications in Pervasive Computing Environments*

Andreas Ulbrich and Torben Weis
Berlin University of Technology
KBS/EN6, Einsteinufer 17
10587 Berlin, Germany
{ulbi,weis}@ivs.tu-berlin.de

Kurt Geihs
University of Kassel
Wilhelmshöher Allee 73
34121 Kassel, Germany
geihs@uni-kassel.de

Abstract

The wide-spread availability of programmable network-enabled embedded devices, such as smart phones, electronic toys, home entertainment systems, and even sensor networks provides an opportunity for many new and exciting applications. However, such environments impose a software engineering challenge due to their inherent heterogeneity and distribution. Model-driven development is a promising approach to deal with such challenges. In this paper we present a graphical modeling language for application development in pervasive computing environments. The language was designed to support the construction of executable application models. It provides simple yet powerful constructs to master common programming task of applications in such environments.

1 Introduction

In this paper we present VRDK, our approach to support application development for pervasive computing environments. VRDK is built around a graphical programming language and is based on the concept of model-driven development.

Model-driven development has been successfully applied to various application scenarios ranging from enterprise applications [19] to combat field technology [8]. These applications assume that the target hardware, i.e. a PC or a server, is powerful enough to run a middleware such as CORBA CCM, or a realtime CORBA implementation. However, this assumption no longer holds for pervasive computing. Such environments comprise small programmable network-enabled devices. Smart phones, mobile robots,

*The work presented in this paper is partially funded by Microsoft Research, Deutsche Telekom Stiftung, and Deutsche Telekom.

game consoles, home entertainment equipment, and even tiny sensor board are able to communicate over wireless networks. These devices impose a software engineering challenge [1]. In comparison to PC and server hardware, issues such as power consumption, CPU speed, and memory capacity become a primary concern.

VRDK offers an executable modeling language for pervasive computing. The main objective of VRDK is to provide a programming abstraction that hides the gore detail from the developer. Developers can create applications in the problem domain. Applications combine available components and react to events. A component can either be a device such as a TV or door bell or a software service such as a browser or media playback. Events can be emitted by components, e.g. DVD playback started, or sensors, e.g. person entered a room. Functionality can be attached to location, e.g. a room. Processing in such a system is event-driven. An event may activate a functionality which then can emit further events, do some processing, or even split into other functionalities.

The next section discusses general aspects of programming models for pervasive computing. Then, we provide an overview of our modeling language and reason on its design. We briefly discuss model execution with VRDK. After that, we present the related work. The paper closes with a discussion and an outlook to future work.

2 Programming Model

Small devices are not small PCs. They do not have hard disks, removable media, or large screens (if any). Instead they have little memory (between 4k and 64k bytes), slow processors (often 8bit), and energy constraints. Applications for small devices are conceptually different from those found on PCs and servers. Most of the devices have some kind of sensors, for example they can measure temperature, detect movement, or collect user input. Furthermore, some have actuators. For example, an remote-controllable light bulb can be treated as an actuator. Therefore, typical applications perform four different actions in an endless loop: sensing, calculating, controlling actuators, sending/receiving messages over the network.

Despite these general similarities, the programming model varies tremendously between different devices. For example, applications for embedded sensors boards (ESB [15]) are basically big loops. During each iteration the sensor values are read and the actuators are set. This iteration must be repeated at least every 150msec otherwise the board resets. In contrast, Windows CE devices execute a full-featured operating system that decouples the application from hardware-specific issues such as polling intervals. Implementing the same behavior once for an ESB and once for a Windows CE device requires writing two totally different implementations because of the different programming models.

2.1 Enforcing a Common Programming Model

Our goal is to provide the same programming model for all devices. This is usually achieved by abstracting from the various combinations of operating system, programming language, virtual machine, and communication system. Through this abstraction a developer perceives a (virtually) homogeneous environment. However, runtime abstractions such as middleware trade off resource consumption for a high-level programming model. Consequently, CORBA is not common on small networked devices and .NET Remoting is missing in the .NET Compact Framework (a subset of .NET for PDAs and embedded devices). An approach that tries to maintain a common high-level abstraction at runtime with a one-serves-all middleware layer is doomed. Many devices lack the capabilities to host such a middleware. Especially very tiny devices have rather restricted programming model. This makes runtime abstractions more costly as they require more CPU cycles and in turn energy, often the most limiting resource.

We conclude that we need high-level programming abstraction at design time. A model-driven approach can provide the desired abstraction and at the same time synthesize implementations suitable for resource-constrained devices.

We provide a modeling language for creating executable application models for pervasive programming environments. This high-level language provides the desired abstraction for such applications by hiding details of target hardware and programming models. Following the model-driven approach, our tools synthesize implementations tailored for the small devices. This does not mean to abandon middleware. If, for example, the target device provides a complete Java runtime then RMI can be utilized. Otherwise, generated code fills in the gaps.

2.2 Parallelism and Distribution

Ubiquitous applications are inherently distributed. A ubiquitous application relies on computing hardware attached to locations (room, floor, etc.) and mobile devices (smartphones, wearable computers etc.) that move with the user. Pervasive computing environments are highly parallel. Sensing devices continuously read sensor values and emit events when the values change. In other event-driven application domains (e.g. graphical user interfaces) it is easy to sequentialize the events. Due to the distribution this is not possible in ubiquitous systems. Thus we put a strong focus on parallelism and distribution in our modeling language. Concurrent processes are first class entities. Communication between processes is asynchronous and event-based. Our modeling language does not provide the concept of global state. Hence, the only way for one process to influence another one is by sending a message. This avoids many problems that could result in unpredictable side effects. Furthermore, our modeling language provides few but powerful elements to handle events from distributed sources.

The high degree of distribution is one of the main challenges of pervasive computing system. With our modeling language the developer can program against the system as a hole. It is not necessary to write application components for different devices. The modeling language exposes an abstract, holistic view of the system, which provides distribution transparency on the modeling level. It is the task of the model transformation tool to partition the application logic and assign the partitions to available devices. Furthermore, the model transformer must generate networking code that glues together the distributed application parts.

3 Application Modeling Language

Our modeling language builds on a small set of core concepts: *components*, *behaviour*, and *location*.

A component represents a piece of hard- of software that offers some well defined functionality. Every component has a location in the physical world. Developers can import pre-fabricated components in VRDK and *use* them in a model. By intention our tool does not provide means for *defining new components*. The main benefit of ubiquitous applications is to tie together functionality that is already available. This idea is fundamental to our modeling language. If you want to develop a new component (e.g. speech recognizer, motion tracker, etc.) you are probably better off using object-oriented modeling and development tools available today. Our modeling language is a domain specific modeling language. It is not a general purpose modeling language like UML. In the following sections we discuss the core concepts of our language in detail.

3.1 Components

Applications embrace functionality of a large range of components, i.e. devices and services. The developer imports components by simply dragging it from a tool bar on the editor window (see Figure 1).

Our modeling tool has no built-in support for any component. Instead, components are added to the tool with plugins. Currently, there are plugins for using Smartphones, PDAs, Media Center PCs, Browser, Media Player, Microsoft Agents, Embedded Sensor Boards (ESB), and RC5 remote controls as components. A component introduces a set of events and commands. The plugin can provide additional types as well as a graphical notation for each command. For example, a Media Center PC emits an

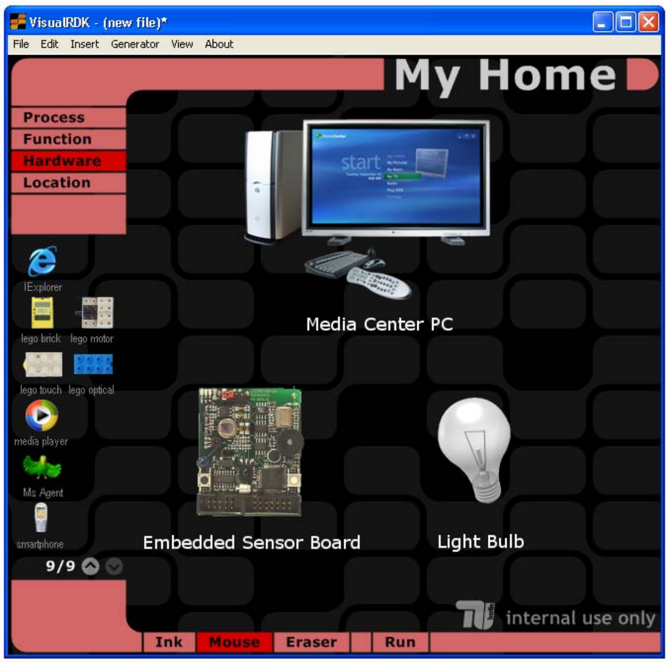


Figure 1: Configuring the available components

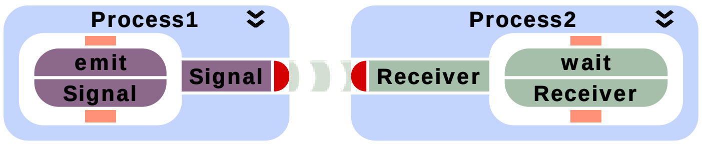


Figure 2: Processes and interprocess communication

event when it starts DVD playback. Furthermore, this component introduces commands for controlling the Media Center, e.g. to start playback or adjust the volume.

3.2 Behavioral Model

Our modeling language builds on the concept of concurrent processes and inter-process communication. Figure 2 shows the notation of two processes. `Process1` defines a signal titled `Signal1`. `Process2` has a signal receiver titled `Receiver1`. An animated conveyor belt can connect signals with receivers. In this example, `Signal` and `Receiver` are connected. The animation of the conveyor belt shows a flow from signal sender to receiver. In order not to obfuscate the diagram permanently, the conveyor belt is only shown if the mouse hovers over the sender or receiver. Its layout is determined automatically.

A process contains a control flow. Control flows orchestrate events and

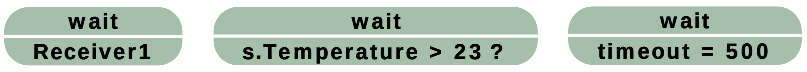


Figure 3: Waiting for events

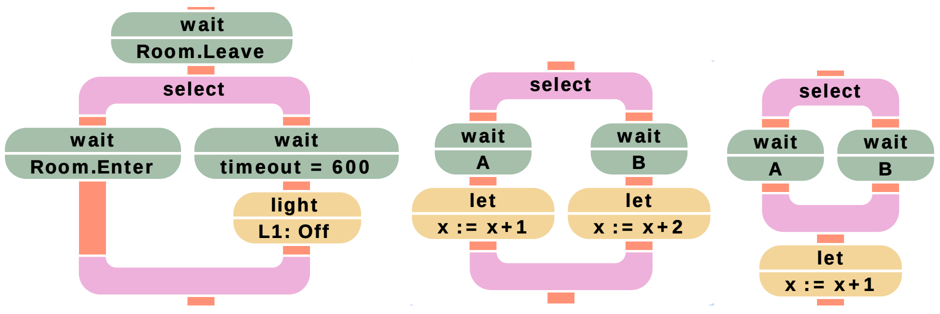
actions. Commands execute actions, e.g. start DVD playback. They are the only way to create side effects in the application. Most commands are introduced by components as discussed above, but some fundamental commands are defined in our modeling language. The `let` command assigns the value of an expression (see section 3.3) to a local variable. Variables do not have to be declared. For ease of use, they are implicitly defined once they are used for the first time.

Applications in ubiquitous computing environments are by their very nature event-driven. Events reflect changes in the environment, for example a sensor value crosses a threshold or a user enters a room. Components define the types of events that are available to an application. Furthermore, the reception of a signal constitutes an event. The `emit` statement emits a signal on a sender (`Process1` in Figure 2). All connected receivers receive the signal.

Our modeling language provides few but powerful constructs to orchestrate events and commands in a control flow. The `wait` statement waits for a single event to occur (Figure 3). It specifies the desired event. A special event is the timeout event. Events are typed, especially they are immutable value types. The event type defines the data that describes the event as properties, e.g. a sensor value or a time stamp. Developers can constrain events with expressions on these properties, for example that the sensor value is above a certain threshold. In this case, the `wait` statement is satisfied only when the event occurred *and* the expression holds.

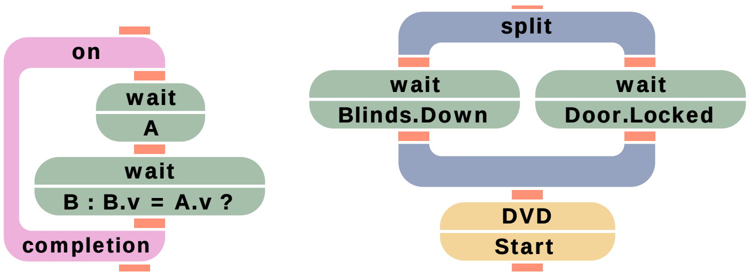
The `select` statement (Figure 4) waits for any of its `wait` statements to complete. It implies an *or*-semantic. Depending on the event, `select` takes a different branch. When both events occur, the one detected first wins. A common usage pattern for `select` is to wait for several events or a timeout as the timeout case often requires additional actions. The introduction of `select` as first class citizen in the language has several advantages. First, it provides an easy to use abstraction for a common programming pattern. Second, it gives the model transformation a high level view on the semantics of the application. Consider the example in Figure 4b/c. It is clear that in (b) the order of event delivery is important to the outcome of the application. In (c), delivery order does not matter. Guaranteeing message ordering in a distributed system is difficult and requires expensive mechanisms. Thus, application level knowledge can be used to synthesize more efficient implementations by selecting the most appropriate ordering mechanism.

An event sequence can be detected by sequentially detecting the events



(a) with timeout (b) order relevant (c) order irrelevant

Figure 4: The `select` statement

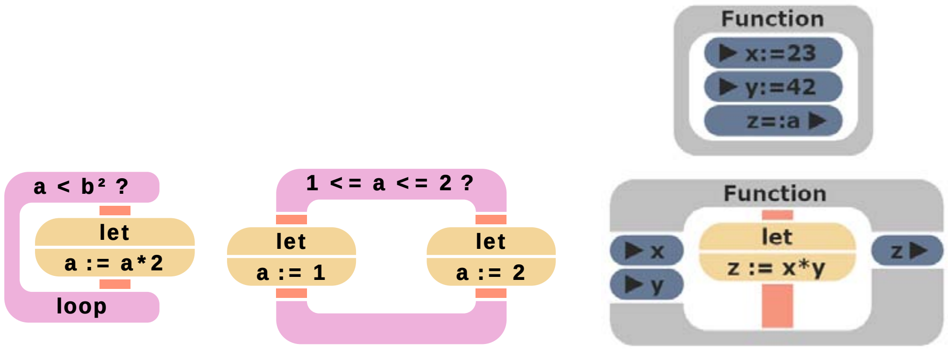


(a) guarded sequence (b) parallel detection

Figure 5: Event sequence and event conjunction

that make up the sequence. The example in Figure 4a detects that a person left a room but did not reenter it in a certain amount of time. In that case the light in the room is switched off.

This simple form of sequencing has a caveat, when a later `wait` is constrained by an earlier event in the sequence. Consider the example in Figure 5a. The first `wait` detects any event A. Once the first `wait` completed, this event is fixed. The second event B is constrained to have the same value as A. As the first A is fixed immediately, the second `wait` will never complete on the event sequence $A(v = 1), A(v = 2), B(v = 2)$. However, in some cases it is necessary to detect $A(v = 2), B(v = 2)$ disregarding of $A(v = 1)$. Thus, the language provides the `on completion` guard for sequences. A guarded sequence is only complete when its last event is detected. The semantics of the guarded sequence are quite different from simple sequences because multiple sequences, e.g. one starting with $A(v = 1)$ and one with $A(v = 2)$, must be detected in parallel as any of them could complete. The guarded sequences completes as soon as the first complete sequence is detected. This concurrent detection implies that a guarded sequence cannot contain commands that change application state as the developer cannot know how often such a command would be executed.



(a) loop and branch (b) function call and definition

Figure 6: Basic control flow elements

The `split` statement executes threads in parallel. It implies a *join* at the end, i.e. it only completes when all its branches complete. When waits are used inside more than one branch of `split`, this statement implies an *and* semantic for events. This is for example useful to synchronize on the completion of more than one activity. The particular order of events is not important. Using combinations of `wait`, `select`, and `split`, the application can wait for every possible conjunction and disjunction of events, conditions, and timeout. In summary, our modeling language provides very easy to use constructs for composite event detection that have the same expressive power as specialized composite event languages [14].

Our compact imperative language does of course support loops and branches (Figure 6a). Loops corresponds to `while`-loops in C-style programming languages. The branch on the right side of Figure 6a tests whether a is in a given range.

For convenience and reuse, developers can define and call functions (Fig. 6b). Functions can have multiple in and out parameters and local variables. No static variables are allowed as this would contradict our requirement that there is no global state.

Figure 7 shows a complete example application. The `DVD` processes switches the room to cinema mode (lights off, high volume) when DVD playback starts and switches back to normal mode when the playback stopped. The second process `Light` switches the light on when someone moves in the room. The special signal `Suspend` and `Reset` are used to coordinate both processes. The `DVD` process suspends `Light` when it is in cinema mode.

3.3 Mathematical Expressions

Our modeling language features side-effect free mathematical expressions. Every command can contain various mathematical expressions to compute values. Thus, the modeling language distinguishes explicitly between com-

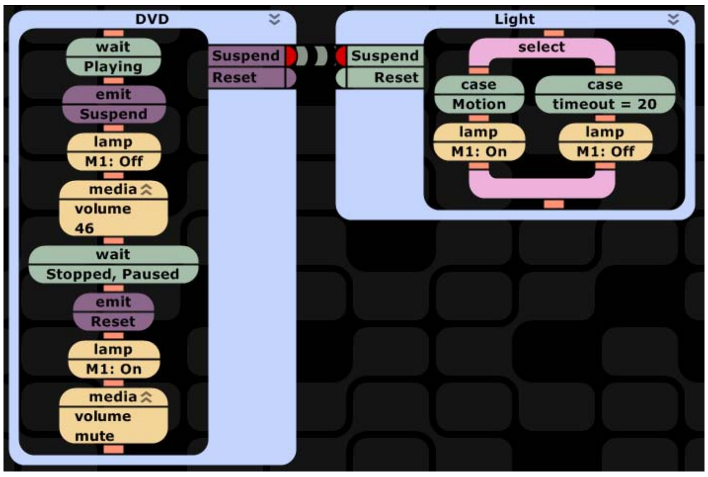


Figure 7: Example application

mands, which produce side-effects, and mathematical computations that are free of side-effects. The main rational behind this restriction is that one expression can be evaluated multiple times at different locations without effecting the applications control flow. This is important for generating a distributed implementation of the application. Imagine a `wait` statement that waits for any temperature sensor to cross a certain threshold. Instead of sending all events from each sensor to a central controller that evaluates the the threshold expression, it is desirable to evaluate this expression on each sensor device and only send a message when the expression holds.

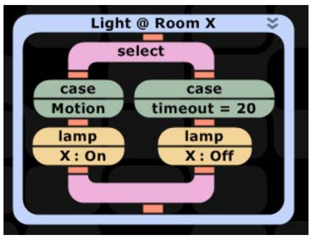
3.4 Location

Location is an essential concept in ubiquitous applications. In VRDK the user arranges components on a floor plan to tell the tool about the location of a component. Thus, the user can create a static *location model* [3][6][12].

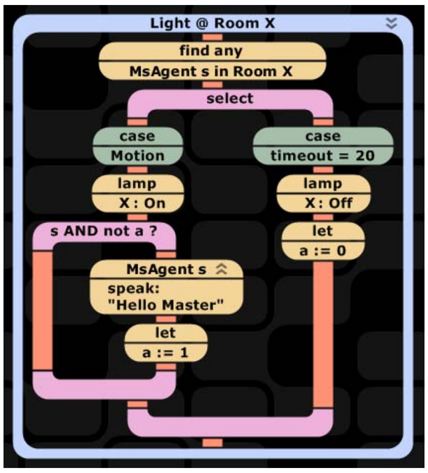
The programmer can either hardwire a location in his script or he can handle location as a parameter. Hardwiring is the straight forward solution to many simple problems. When you want to play music in the living room, you can simply instruct the "play" command to use the Media Center PC, which is located in the living room.

Now imagine you want to write a script that turns on the light in *any* room when someone enters the room. Using hardwired locations, the user would have to program one process for every room. This results in duplicated code since the processes will all contain the same control flow. The only differences are the hardwired devices.

In our modeling language we solve this problem with location-dependent *process groups* (Figure 8a). The process group executes one process for every location, for example for every room, floor, or house. All processes of



(a) process group



(b) location model query

Figure 8: Location-dependent processing

the group share the same control flow. Creating such a process group is extremely simple. The developer just selects `Run in every room` in the property dialog of the process. Instead of using location-specific components or services (e.g. the motion sensor in the living room), the user can use anonymous components. For process groups, VRDK offers an anonymous instance of every supported component or service. Upon execution of a process group, VRDK detects which anonymous components are used and determines via the location model their location-specific counterparts. In our example, VRDK will find out that the process group requires one motion sensor and one controllable lamp per room. Rooms that do not feature the required components do not participate in the process group.

Another way of dealing with location in VRDK is to query the location model at runtime. Using the `find any` command the developer can search for a certain device type at some location. Figure 8b uses command to extend the previous example. If someone enters the room, the application searches for an MsAgent and lets it speak. VRDK detects the query for this component type and does therefore not demand such a component for every room. Hence, we can still use one process group even though not every room has an MsAgent.

4 Model Execution

VRDK has a built-in interpreter that can directly execute the model. The first step of the interpreter is to build a state machine for every process. In the second step, the interpreter initializes all components specified in the model. Therefore, it uses the plugins. A VRDK plugin support the interpreter with components initialization, execution of component commands, and notification of component events.

The interpreter is especially useful for rapid prototyping since it allows the developer to run and test a complex distributed application by just hitting the `Run` button. This convenience comes at a cost. Most notably, the interpreter executes on the same PC that runs VRDK. All components are remote controlled by this PC and must keep a connection to this PC. Especially in the context of mobile devices such as PDAs or Smartphones, this is no viable solution for a productive environment. Furthermore, the network load is tremendous since every piece of information must be sent to the PC. However, the interpreter is very useful for testing and debugging.

One common problem of debugging an application developed with a model-driven tool chain is the missing abstraction at debugging time. Usually, debuggers operate on the PSM-level (i.e. C++ or C#) while the programmer would prefer working on the PIM-level. Using the interpreter this problem is eliminated. The interpreter knows which process and command of the model it currently executes. Thus, it can provide the developer feedback directly on the PIM-level instead of the PSM-level.

For the deployment of the application in a real system the model must be executed by a distributed implementation. It is the task of model transformers to synthesize these implementations. The model transformers partition the application model and assign application parts to different devices. Our transformers work on the same state machines that are used for the interpreter. State machines can be distributed over a set of devices. In order to make a state transition, an activation message must be sent to the device that has the target state. Generating efficient and robust distributed implementations is still subject of ongoing research. The discussion of code generation is beyond the scope of this paper.

5 Related Work

The need for programming abstractions for pervasive computing has led to a number of different solutions. As already discussed before, runtime abstractions are one way to approach the problem. One example is BASE [5] and its complementing component model PCOM [4], both run on top of the J2ME Limited Connected Device Configuration [16]. While BASE/PCOM offers a fairly sophisticated programming model, BASE/PCOM components are written against a device and the developer is responsible to orchestrate the communication between components on different devices. At the same time J2ME prohibits the use on very small devices.

Approaches like TinyOS [10] target even smaller devices. However, TinyOS provides an even less powerful programming model. On interesting aspect of TinyOS and TinyOS applications is that they are implemented in nesC [7]. The language is an extension of C that restricts the expressiveness of C in such a way that full program analysis is possible. This can be used for dead-code elimination or to detect race conditions. This is useful for embedded devices and shares some similarities with a model-driven approach.

Model-driven development is a viable approach to deal with the complexity in certain application domains [19]. Consequently, it has been applied to the field of distributed embedded systems. For some problem domains standardized extensions to the UML exist, e.g. the UML Real-time profile. Tools like AIRES analyze models with real-time constraints and synthesize an implementation [9]. The CoSMIC [8] tool applies model-driven development to the field of distributed embedded real-time systems. As a main difference to our approach CoSMIC assumes the existence of a rich execution platform such as CIAO [17]. The model transformation main responsibility is to configure the middleware for QoS provisioning. Our first approach was to build on our extensible UML-based modeling tool Kase [18]. However, we came to the conclusion that UML is not an ideal basis for our modeling language. UML originates in the object-oriented paradigm that is well suited for dealing with complexity of application logic. However, in our application domain the application logic is rather simple compared to its implementation. As UML is a general purpose modeling language, it does not feature the modeling concepts that we need. But extending and redefining the semantics of UML often leads to undesirable results [2].

Recently research started to focus on more high-level programming abstractions. Location-enhanced applications are a special application domain for pervasive computing. Topiary [13] is a graphical tool for prototyping location-enhanced applications. Unlike our approach it is built on the idea of storyboards. Developers can define what GUI elements are shown when certain constraints on the locations of users are met. However, due to the storyboard the control flow is limited and it was not designed to actually control the environment.

In [11] an editor for applications in ubiquitous domestic environments is presented. It allows the user to connect components using a jigsaw puzzle-style approach. Devices available in a room present themselves as a jigsaw piece on a graphical editor. For example, pieces for motion detector, camera, and PDA can be combined to construct a simplistic surveillance system. The motion detector triggers the camera and the camera shows its picture on the PDA. The drawback of this approach is that it has only a linear control flow and no means for mathematics. Essential control structures like branches and loops cannot be mapped to this programming model.

6 Conclusions & Outlook

We presented a modeling language for applications in pervasive computing environments. Our executable modeling language is specialized for this application domain. Parallelism, events, location, and components are first class citizens in the language. With the interpreter we could test the semantics of the language and the usefulness of the modeling elements. We believe that only with high level abstractions and a careful restriction of expressiveness, i.e. no global state, no side-effects, the generation of efficient and robust implementations is possible. A model-driven approach has substantial advantages over a solely source-code-based development process. It maintains a high-level abstraction and can at the same time target heterogeneous devices with different programming models.

Currently our research focuses on the generation of implementations. Our two main concerns are efficiency and robustness. Pervasive computing systems are prone to errors. Messages can get lost, devices can fail, and even worse the user can interfere with the environment in any unforeseeable way. Applications should keep working in face of such errors or recover automatically. Due to resource restrictions classic fault-tolerance techniques are not suitable. Thus, we investigate how self-stabilizing implementations can be generated.

References

[1] G. D. Abowd. Software engineering issues for ubiquitous computing. In *Proceedings of The 21st International Conference on Software Engineering (ICSE 1999)*, page 75, Los Angeles, USA, May 1999. IEEE.

[2] C. Atkinson, T. Kühne, and B. Henderson-Sellers. Stereotypical encounters of the third kind. In *UML 2002*, 2002.

[3] C. Becker and F. Dürr. On location models for ubiquitous computing. *Personal and Ubiquitous Computing*, 9(1):20–31, 2005.

[4] C. Becker, M. Handte, G. Schiele, and K. Rothermel. PCOM – A component system for pervasive computing. In *Proceedings of the Second International Conference on Pervasive Computing and Communications (PerCom'04)*, page 67, Orlando, Florida, 2004.

[5] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel. BASE – A micro-broker-based middleware for pervasive computing. In *Proceedings of the First International Conference on Pervasive Computing and Communications (PerCom'03)*, page 443, Fort Worth, Texas, 2003.

[6] B. Brumitt and S. S. Topological world modeling using semantic spaces. In *Workshop on Location Modeling for Ubiquitous Computing*, 2001.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11. ACM Press, 2003.

[8] A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt. Model Driven Middleware: A new paradigm for deploying and provisioning distributed real-time and embedded applications. *Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2004.

[9] Z. Gu and K. G. Shin. Synthesis of real-time implementation from uml-rt models. In *Proceeding of the RTAS Workshop an Model-Driven Embedded Systems (MoDES'04)*, Toronto, Canada, May 2004. IEEE.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System archtecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[11] J. Humble, A. Crabtree, T. Hemmings, K. kesson, B. Koleva, T. Rodden, and P. Hansson. playing with the bits user-configuration of ubiquitous domestic environments. In *UbiComp2003*, 2003.

[12] B. Johanson, A. Fox, and T. Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1(2):67–74, 2002.

[13] Y. Li, J. Jong, and J. Landay. Topiary: a tool for prototyping location-enhanced applications. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 217 – 226. ACM Press, 2004.

[14] P. R. Pietzuch, B. Shand, and J. Beacon. Composite event detection as a generic middleware extension. *IEEE Network*, pages 44–55, January/February 2004.

[15] J. Schiller, A. Liers, H. Ritter, R. Winter, and T. Voigt. Scatterweb - low power sensor nodes and energy aware routing. In *Proceedings of Hawaii International Conference on System Sciences (HICSS 2005)*, Hawaii, USA, Jan. 2005.

[16] Sun Microsystems, Inc. J2ME Connected Limited Device Configuration (CLDC), Specification Version 1.1. JSR-139, Mar. 2003.

[17] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill. *QoS-enabled Middleware*. Wiley and Sons, New York, 2003.

[18] T. Weis, A. Ulbrich, and K. Geihs. Model metamorphosis. *IEEE Software*, 20(5):46–51, September/October 2003.

[19] T. Weis, A. Ulbrich, K. Geihs, and C. Becker. Quality of service in middleware and applications: A model-driven approach. In *Proceedings of the 8th International Enterprise Distributed Object Computing Conference (EDOC 2004)*, Monterey, California, USA, Sept. 2004. IEEE CS Press.

ISBN 952-12-1556-9
ISSN 1239-1905