Towards a formal specification of an electronic payment system in CSP-CASL

Andy Gimblett[1], Markus Roggenbach[1], and Bernd-Holger Schlingloff[2]

[1] University of Wales Swansea, Department of Computer Science, United Kingdom
[2] Fraunhofer Institute FIRST and Humboldt University at Berlin, Germany

Abstract. This paper describes the formal specification of a future banking system by abstract data types and process algebra. In contrast to previous exercises (e.g., [1]), the system's description is an actual industrial standard which is being used to develop the next generation of automatic banking machines. The specification language CSP-CASL is particularly well suited to this type of problem, since it combines both control and data aspects and allows loose specification of data types for later refinement. During the formalisation, several inconsistencies and ambiguities were exhibited. The obtained specification serves as a starting point for further validation.

1 Introduction

Electronic payment systems represent an important application area for both the theory and practice of system specification. In theory, they provide a suitable benchmark to demonstrate the abilities of a certain specification method (consider e.g. [1, 5, 7]). In practice, they are classified as safety critical systems and thus must be developed with due diligence. In this paper we consider such an application by studying in detail how to build a formal specification for the electronic payment system ep2 [2], a new international standard developed by a consortium of leading Swiss finance institutes.

ep2 is typical of a number of similar applications. The system consists of seven autonomous entities centred around the ep2 *Terminal*: Cardholder (i.e., customer), Point of Service (i.e., cash register), Attendant, POS Management System, Acquirer, Service Center, and Card, see Fig. 1. These entities communicate with the Terminal and, to a certain extent, with one another via *XML-messages* in a fixed format. These messages contain information about authorisation, financial transactions, as well as initialisation and status data. The state of each component heavily depends on the content of the exchanged data. Each component is a *reactive system* defined by a number of *use cases*. Thus, there are both reactive parts and data parts which need to be modelled, and these parts are heavily intertwined.

The ep2 system also represents a typical industrial case study. The specification consists of roughly 600 pages of text, which is a mixture of plain English and other semi-formal notation. Some parts are specified up to a bit encoding level,

while others are left open and referred to common understanding. It is, however, an actual international standard which is used to implement and validate banking machines from different manufacturers.

In the formalisation, we use the specification language CSP-CASL [22]. This language combines process algebraic specification of reactive behaviour and algebraic specification of data types at various levels of detail. CSP-CASL uses the process algebra CSP [10, 23] for the modelling of reactive behaviour, whereas the properties of the communications are specified in CASL [3, 16]. CSP-CASL is generic in the CSP semantics. Furthermore, CSP-CASL offers a notion of refinement with clear relations to both data refinement in CASL and process refinement in CSP.

Structuring our CSP-CASL specifications in nearly the same way as the original ep2 documents allows us to exhibit several ambiguities, omissions, and contradictions in the documents. Here, especially CSP-CASL's loose specification of data types plays an important role. Often, the top level ep2 documents provide only an overview of the data involved, while the presentation of further details for a specific type is delayed to separate low-level documents. CSP-CASL is able to match such a document structure by a library of specifications, where the informal design steps of the ep2 specification are mirrored in terms of a formal refinement relation.

The paper is structured as follows. First, we give an overview of the ep2 system, where we focus on the existing specification and the shortcomings thereof. Then, we quickly review the specification language CSP-CASL. In section 3, we describe our formalization, and in section 4 we report on our results with this formalization. Finally, we summarize our results, discuss related approaches, and conlude with hints on future work and perspectives.

2 The ep2 system

ep2 stands for 'EFT/POS 2000', short for 'Electronic Fund Transfer/Point Of Service 2000', and is a joint project established by a number of (mainly Swiss) financial institutes and companies in order to define EFT/POS infrastructure for credit, debit, and electronic purse terminals in Switzerland (www.eftpos2000.ch). ep2 builds on a number of other standards, most notably EMV 2000 (the Europay/Mastercard/Visa Integrated Circuit Card standard, see www.emvco.com) and various ISO standards. An overview of ep2 is shown in Fig 1.

2.1 ep2 document structure

The ep2 specification consists of twelve documents, each of which either considers some particular component of the system in detail, or considers some aspect common to many or all components. The Terminal, Acquirer, POS Management System, Point of Service (POS), and Service Center components all have specification documents setting out 'general', 'functional', and 'supplementary'

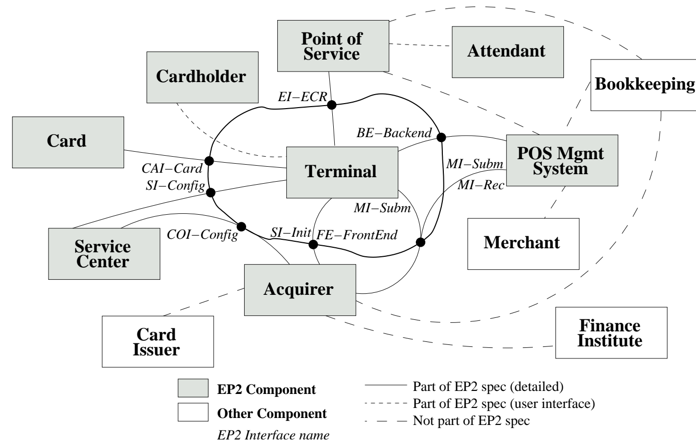

Fig. 1. Overview of the ep2 System, following closely [2].

requirements, where the functional requirements carry the most detail, and consist mainly of use cases discussing how that particular component behaves in various situations. As well as the specifications of particular components, there is a Security Specification, an Interface Specification, and a Data Dictionary.

One obvious characteristic of such a document structure is that, when considering some aspect of the system, the information required to understand that aspect is contained in several different documents, each of which has its own things to say about the situation in question. For example, in order to gather all information about the SI-Init interface between Terminal and Acquirer, see Fig. 1, one has to examine the Terminal Specification, the Acquirer Specification, the Interface Specification, and the Data Dictionary. As we will see, this approach easily leads to inconsistencies and ambiguities.

2.2 ep2 specification style

The original ep2 documents are comprised of a number of different specification notations: plain English; UML-like graphics (use cases, activity diagrams, message sequence charts, class models, etc.); pictures; tables; lists; file descriptions; encoding rules.

Subsequently, we will focus on the *SI-Init* connection between *Terminal* and *Acquirer* (see Fig. 1).
The Acquirer is defined in a table of roles as a *"Card processor, which runs a system for processing of electronic payment transactions. The Acquirer is in contact with the merchant."* Later, in another table describing the main system features, the functionality of the Acquirer is classified into four subsystems:

- *Acquirer Initialisation System:* Supports remote SW-parameter initialisation. Exchanges Terminal configuration data with the Service Center.

– *Authorisation System:* Processes Terminal on-line authorisation requests, as well as transaction reversal requests. Forwards issuer scripts to the Terminal.
– *Submission System:* Processes transactions.
– *Reconciliation System:* Provides reconciliation[1] data to the merchant.

In the Acquirer general requirements document, a fifth subsystem is identified:

– *COI* [2] *server:* Used for data exchange with the Service Center.

Another table lists the communication interfaces; in particular, *"The SI-Init interface is used by the Acquirer to download application specific initialisation data which include Acquirer data necessary for Acquirer authentication and data submission."*

Later in the System specification, this communication is depicted in a use case, seen in Fig 2. It shows that the "Get Initialization Function" can be called by the service man either directly at the Terminal, or via an "Initiate Terminal Setup" at the Point of Service. Additionally, the function can be called in cyclic intervals by a timer process, or by an authentication server process at the Acquirer's site.

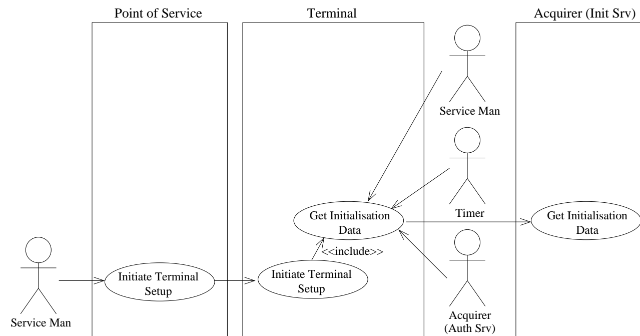

Fig. 2. Part of use case for Get Initialisation Function as shown in [2].

For both the Terminal and the Acquirer, activity diagrams are given describing the flow of control on the receipt of messages. For conciseness, in Fig. 3 we only show the diagram for the Acquirer.

For each state in this activity diagram, a verbal description is given of which message parameters are admissible in this state, and what the appropriate answer messages are composed of. For example, in state "Send ≪Config Data Request≫ Message":

---

[1] Reconciliation: to compare the business undertaken at the terminal with that recorded by the acquirer and credited to the merchant's bank account.

[2] COI stands for *configuration and initialisation* (of the terminal) within the ep2 specification.

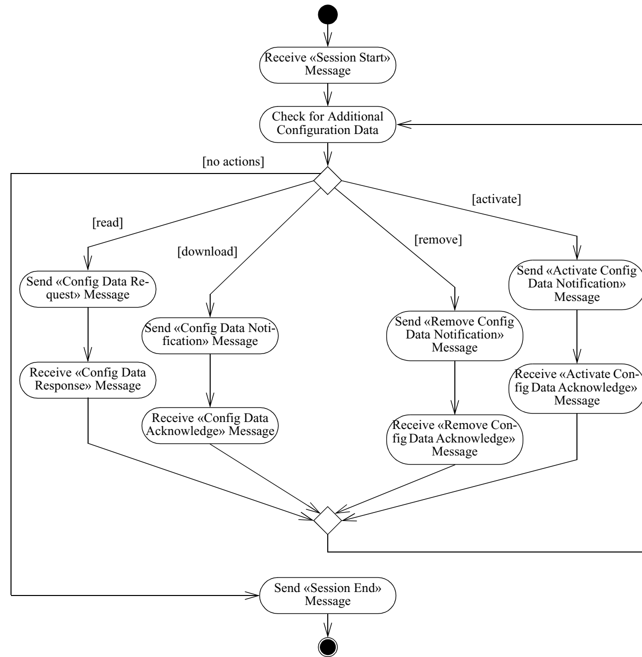

Fig. 3. Activity diagram for Acquirer getting initialisation data, as shown in [2].

> *The Acquirer shall send the message ≪Config Data Request≫ to the Terminal. The Acquirer shall set <Config Data Object> to the configuration data object which the Acquirer is interested in. For CPTD, TACD and CAD the Acquirer shall specify with an AID resp. a RID which table exactly it wants to receive. If the Acquirer sets <Config Data Object> to LAID, he receives a list of all AID's supported by him from the Terminal.*
> ...

The appropriate parameter values are informally described in another table, the beginning of which is given in Fig. 4.

On the concrete data encoding level, the SI-Init connection is constrained by the following requirements in the system description:

- ep2 interface.
- Uses XML based on TCP/IP.
- Message based.
- Uses strong security mechanisms.

2.3 Shortcomings

The above specification style is typical for a number of today's industrial developments. As described above, it uses a number of up-to-date specification

<Config Data Object> value	Object Name	Additional Data Element	Returned by Terminal
ACD	Acquirer Config Data	-	One ACD object of the requesting acquirer
AISD	Acquirer Init Srv Data	-	One AISD object of the requesting acquirer
CPTD	Card Profile Table Data	<Application Identifier (AID)>	One CPTD object of the requested AID.
CAD	Certification Auth Data	<Registered Application Provider Identifier (RID)>	One CAD object of the requested RID.
TACD	Terminal Application Config Data	<Application Identifier (AID)>	One TACD object of the requested AID.
...	...	...	...

Fig. 4. Excerpt of message parameters and expected answers for initialisation [2].

notations, and has additional verbal explanations and cross references throughout the books. However, for a team of developers which has to rely on this specification as a sole basis for an implementation it may be hard to produce a correct result. (A typical scenario would be a company which is not part of the consortium and wishes to produce a compliant device). Some of the reasons for this are:

First, there are several *ambiguities* within the documents which could lead to interoperability problems between different implementations. Ambiguities are inherent in all natural-language documents, since human language is subject to individual interpretation. As an example, consider the expected answer *"One ACD object of the requesting Acquirer"* in Fig. 4. This could mean

– One object, and it must be the one of the requesting Acquirer.
– One of all the objects belonging to the requesting Acquirer.

(In mathematical logic the difference is formalized by Russell's jota- and Hilbert's eta-operators.) Different opinions about the meaning of this requirement could lead to incompatible implementations.

Worse, there are some *inconsistencies* within the documents themselves. In fact, the data flow for the "Acquirer Init Srv Data" message is specified in the data dictionary as *from* the Acquirer via Service Center *into* the Terminal. In the above activity diagram, the Acquirer is allowed to *read* this data object *from* the Terminal. It contains the Acquirer's identifier, public key and communication address. The only plausible reason for the Acquirer to receive this data is to check its consistency. However, the Acquirer has no way to initiate a correction of these data, even if an inconsistency is detected. Since the specification is rather large and was written by several authors, such situations cannot be avoided.

Third, the ep2 documents are not suitable for *tool supported software development*. In particular, since the various requirements are intermingled, they cannot be easily input into an automated requirement management system such

as Telelogic's DOORS or IBM/Rational's Requisite Pro. Thus, it is hard to assure that all required functionality is present in an implementation. Moreover, it is not possible to automatically check consistency of the requirements with one another, or to prove the conformance of a particular implementation with respect to the specification.

Last but not least, the given documents *interleave different levels of abstraction*. For example, the above mentioned architecture of the Acquirer is augmented by "logical component requirements" such as permanent accessibility, as well as use cases and a data model. Thus, it is not easy to use the specification in a structured development process. In fact, since implementation details are to be found throughout the specification, a programmer might be forced to reinvent parts which have already been developed by others. Moreover, implementation details are often subject to change; thus, the whole ep2 specification must be updated whenever some detail is modified. This can result in serious version compatibility problems.

3 CSP-CASL

CSP-CASL [22] is a comprehensive language which combines the specification of *data types* in CASL [3, 16] with *processes* written in CSP [10, 23]. The general idea of this language combination is to describe reactive systems in the form of processes based on CSP operators, but where the communications between these processes are the values of data types, which are loosely specified in CASL. All standard CASL features are available for the specification of these data types, namely many-sorted FOL with sort-generation constraints, partiality, and subsorting[3]. Furthermore, the various CASL structuring constructs can be used to describe data types within CSP-CASL. This includes the structured **free** construct, which adds the possibility to specify data types with initial semantics. For the description of processes, the typical CSP operators are included in CSP-CASL: there are for instance internal choice and external choice; the various parallel operators like the interleaving operator, the alphabetized parallel operator, and the general parallel operator; also communication over channels is included. Similarly to CASL, CSP-CASL specifications can be organized in libraries. Indeed, it is possible to mix CASL specifications and CSP-CASL specifications in one library, separating the development of data types in CASL from their use within CSP-CASL. This allows the specification of a complex system like ep2 in a modular way.

Syntactically, a CSP-CASL specification with name N consists of a data part Sp, which is a structured CASL specification, an (optional) channel part Ch to declare channels, which are typed according to the data specification, and a process part P written in CSP, within which CASL terms are used as communications, CASL sorts denote sets of communications, relational renaming is

[3] For technical reasons, in CSP-CASL sub-sorting is restricted to subsort relations with so-called top elements. As it turns out e.g. in our current case study of specifying ep2, this restriction is of no practical relevance.

described by a binary CASL predicate, and the CSP conditional construct uses CASL formulae as conditions:

$$\textbf{ccspec } N = \textbf{data } Sp \textbf{ channel } Ch \textbf{ process } P \textbf{ end}$$

See Fig. 6 for a concrete instance of such a scheme. In the process part, the `let ... in ...` construct offers the possibility for recursive process definitions. Processes can also be parameterized with variables typed by CASL sorts. In general, this combination of recursion and parameterization leads to an infinite system of process equations. The theory of CSP offers syntactic characterizations for the existence and uniqueness of solutions for such systems of equations.

As a consequence of the loose semantics of CASL, *semantically* a CSP-CASL specification is a family of process denotations for a CSP process, where each model of the data part Sp gives rise to one process denotation. The definition of the language CSP-CASL is generic in the choice of a specific CSP semantics. For example, all denotational CSP models mentioned in [23] are possible parameters. For the purpose of specifying ep2 in CSP-CASL, we mainly use the CSP denotational stable-failures model. This model is able to distinguish between the different choice operators, and allows for infinite non-determinism as well as for infinite communication alphabets: features which naturally appear in abstract system descriptions involving loosely specified data types.

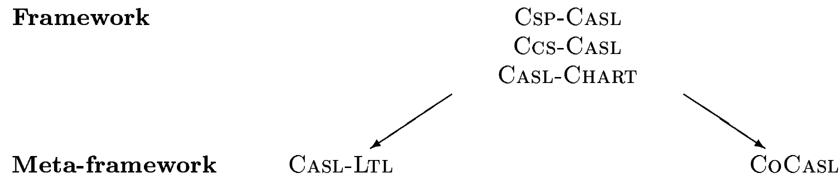

Fig. 5. Relationship between CSP-CASL and other reactive CASL extensions

Related Specification Languages Within the context of CASL, various reactive extensions were proposed – see Figure 5 for a small selection and classification. Our definition of CSP-CASL, like CCS-CASL [24, 25] and CASL-CHART [20], combines CASL with a particular mechanism to describe reactive systems. This results in a *Framework* suitable to model real-world systems. CASL-LTL [19] and COCASL [21, 15], on the other side, can be seen as a *Meta-framework* aiming more for the formalization of (the semantics of) different frameworks for reactive systems.

Outside the CASL context, e.g. μCRL [9], LOTOS [11], and E-LOTOS [13] provide other solutions for the integrated specification of data and processes within one language. Conceptually, μCRL and CSP-CASL are quite similar in their respective design. In the data part however, CSP-CASL is far more rich: among other features, it offers partiality and subsorting which are frequently

used in the modelling of ep2. LOTOS [11] and its recently defined successor E-LOTOS [13] use for data description initial semantics and a functional programming language, respectively. Thus, these languages do not allow for the modelling of the abstract system layers of ep2 as presented here within CSP-CASL.

4 Formalizing ep2 in CSP-CASL

The present formalization of the ep2 system is the first major industrial case study in CSP-CASL. It was done with a number of different aims. Our main objective was to show the feasibility of the approach. This includes many aspects:

Scalability We wanted to show that it is possible to completely specify a non-trivial system in this formalism. Previous approaches restricted themselves to academic toy examples or small fragments of actual systems.

Expressiveness Another aim was to prove that CSP-CASL encompasses enough expressive power to deal with the given application. In particular, ep2 contains most aspects which can be found in typical present-day computational systems.

Usability An important point was to demonstrate that CSP-CASL specifications are easy to write and easy to understand. Many specification formalisms are only targeted at experts and require intensive training and experience.

Adequacy In order to investigate to what extent the informal and natural language descriptions can be formalized, we wanted to follow the original document structure as closely as possible.

A second objective relates to the actual ep2 system itself. We wanted to show how formal methods can help to improve the design.

Clarity By structuring the formal specification appropriately, we wanted to untangle the different levels of abstraction in the documents. This could guide future implementors in building a modular implementation.

Precision We wanted to exhibit ambiguities and inconsistencies within the informal descriptions, which facilitate implementations by third-party implementors.

Validation and Verification In a second step, we want to use the resulting formal specification to validate actual implementations, prove their conformance with the standard and to generate test cases from the formal specification.

In this section, we give an overview on the structuring of our formalization. According to the general paradigm of CSP-CASL, there are two main aspects: the reactive behaviour of ep2 components and the data structures which are involved.

4.1 Reactive behaviour

It is natural to model ep2 as a reactive system. In CSP-CASL, we describe its different components by CSP processes which interchange data over communication channels typed by CASL sorts.

```
1   ccspec ep2 =
2   data sorts D_CAI_Card; D_SI_Config; D_SI_Init; D_FE_FrontEnd;
3              D_MI_Subm; D_BE_BackEnd; D_EI_ECR;  D_COI_Config; D_MI_Rec;
4        free type D_MI_Subm_or_Rec ::=
5            subm(select_subm:? D_MI_Subm) | rec (select_red:?  D_MI_Rec);
6   channels C_CAI_Card: D_CAI_Card;      C_SI_Config: D_SI_Config;
7            C_SI_Init: D_SI_init;        C_FE_FrontEnd;
8            C_MI_Subm: D_MI_Subm;        C_BE_BackEnd: D_BE_BackEnd;
9            C_EI_ECR: D_EI_ECR;          C_COI_Config: D_COI_Config;
10           C_MI_Subm_or_Rec: D_MI_Subm_or_Rec;
11  process
12    let Card          =     Run(C_CAI_Card)
13        ServiceCenter  =     Run(C_SI_Config)  ||| Run(C_COI_Config)
14        Acquirer       =     Run(C_COI_Config) ||| Run(C_SI_Init)
15                         ||| Run(C_FE_FrontEnd) ||| Run(C_MI_Subm)
16                         ||| Run(C_MI_Subm_or_Rec)
17        PosMgmtSystem  =     Run(C_BE_BackEnd) ||| Run(C_MI_Subm_or_Rec)
18        PointOfService =     Run(C_EI_ECR)
19        Terminal       =     Run(C_CAI_Card)   ||| Run(C_SI_Config)
20                         ||| Run(C_SI_Init)    ||| Run(C_FE_FrontEnd)
21                         ||| Run(C_MI_Subm)    ||| Run(C_BE_BackEnd)
22                         ||| Run(C_EI_ECR)
23    in Terminal
24         [| C_CAI_Card, C_SI_Config, C_SI_Init, C_FE_FrontEnd,
25            C_MI_Subm, C_BE_BackEnd, C_EI_ECR |]
26       (Card
27        ||| ((ServiceCenter
28                  [ C_COI_Config || C_COI_Config, C_MI_Subm_or_Rec ]
29              Acquirer)
30                  [ C_COI_Config, C_MI_Subm_or_Rec || C_MI_Subm_or_Rec ]
31              PosMgmtSystem)
32        ||| PointOfService)
33  end
```

Fig. 6. Modelling ep2: The architectural level.

On the *architectural level* in the center of the ep2 system there is a `Terminal` process – c.f. line 23 of Fig. 6. This `Terminal` communicates over channels with its environment, expressed here in terms of the CSP general parallel operator `[| C_CAI_Card, ..., C_EI_ECR |]` linking the `Terminal` with its envi-

ronment. The environment consists of the processes `Card`, `ServiceCenter`, ..., `PointOfService`.

Note how this model directly corresponds to Fig. 1, which is the first and most abstract description of ep2 given in the ep2 System Specification. We express this correspondence by the choice of names: ep2 components become identically named *processes*, an ep2 interface is characterized by the possible data to be exchanged over it — prefix `D_` for the corresponding *sort* providing the type of this data — and by the connection it represents — prefix `C_` for the corresponding *channel*.

We do not model the Cardholder and the Attendant as processes as the ep2 specification covers their role only on the level of user interfaces. Most of the processes in the environment run independently of each other, expressed by the CSP interleaving operator `|||` (lines 27 and 32). Some of them also interchange information which each other: the `ServiceCenter`, the `Acquirer`, and the `PosMgmtSystem`. Here, we use the CSP alphabetized parallel operator, e.g. `[ C_COI-Config || C_COI-Config, C_MI_Subm_or_Rec ]` (line 28), which synchronizes in the intersection of the two alphabets, i.e. in this example in `C_COI-Config`. On the architectural level, we leave the behaviour of the different processes completely unspecified, i.e. they are modelled by the CSP process `RUN(A)`, which is the deadlock-free, non-terminating process able to engage in any event in a set of communications `A`. For any process of the ep2 system we choose this set `A` to consist of all messages, which it might send or receive over the channels which connect it to other processes. For instance, for the `Terminal` the set `A` consists of all messages which can be sent or received over any of the channels named in the general parallel `[|C_CAI_Card, C_SI_Config, ..., C_EI_ECR|]` which connects the `Terminal` with its environment. This is expressed here as the interleaving of several `Run` processes (lines 19 – 22).

On the *abstract component description level*, we refine the processes `RUN(A)` of the above architectural model without changing the overall communication structure. Our example stems from the `Terminal` specification, showing the Terminal's reactions to the Acquirer's requests on initialization data. In a first step, we specify only that the Terminal produces answers of the right kind, e.g. on a `D_SI_ConfigDataRequest` a `D_SI_Init_ConfigDataResponse` is sent:

```
TConfigManagement =  C_SI_Init ? x ->
 if x in D_SI_ConfigDataRequest
   then !y:C_SI_Init.D_SI_Init_ConfigDataResponse -> TConfigManagement
 else if x in D_SI_ConfigDataNotification
   then !y:C_SI_Init.D_SI_Init_ConfigDataAcknowledge -> TConfigManagement
 else ...
```

Here, `!y:A -> P` denotes the process which first communicates a value `y` out of the set `A` and then behaves like `P`; i.e. the `!` operator is similar to the CSP prefix choice, but for the former the choice is internal, while for the latter the choice is external.

In the next step, the *concrete component description level*, we model which specific values the Terminal is going to send. It is at this level, that the process becomes *stateful*, i.e. it depends on a parameter `p:Pair[TState][Trigger]`.

Here, `TState` represents the `Terminal`'s memory, while `Trigger` says what kind of signal initiated the configuration management.

```
TConfigManagement (p:Pair[TState][Trigger]) =  C_SI_Init ? x ->
 if x in D_SI_ConfigDataRequest
   then C_SI_Init ! configDataResponse(x,state(p))
     -> TConfigManagement(p)
 else if x in D_SI_ConfigDataNotification
   then C_SI_or_FE ! configDataAcknowledge
     -> TConfigManagement (pair(activateData(x,state(p)),trigger(p)))
 else ...
```

This example illustrates the interaction between the specification of reactive behaviour and the modelling of data types when studying the control flow within a component: A message `x` is received from the `Acquirer` over the channel `C_SI_Init`. Depending on the type of `x`, different answers are sent back to the `Acquirer`, e.g. information `configDataResponse(x,state(p))` on the current configuration of the Terminal or a message `configDataAcknowledge`. Then the configuration management is continued, either without a state change or with a state change to `pair(activateData(x,state(p)),trigger(p))`.

It is at the component description levels that more information on data in terms of CASL elements come into play: for instance, there is the test if the value `x` belongs to a certain subsort `D_SI_ConfigDataRequest`. The response is computed by a function `configDataResponse` that takes the message `x` and the current state `state(p)` of the `Terminal` as parameters, or the new state is computed by a function `activateData(x,state(p))`.

4.2 Data on different levels of abstraction

In direct correspondence to the development of ep2's reactive behaviour over different levels of abstraction, the data types involved are made more and more concrete.

On the *architectural level*, see Fig. 6, it is sufficient to speak merely about the existence of sets of values which are communicated over channels; e.g. the data `sort D_CAI_Card` is interchanged on a `channel C_CAI_Card: D_CAI_Card` between the `Card` and the `Terminal`. Or a channel shall be shared by different message types, as `channel C_MI_Subm_or_Rec: D_MI_Subm_or_Rec`. Here, the CASL free type construct ensures that the different kinds of data are kept separate.

If the *component specification level* is abstract, it is usually sufficient to introduce suitable subsorts. Consider for instance the communication between Acquirer and Terminal, see Fig. 3. To specify how the `Acquirer` interchanges initialisation data, it is enough to know the type of the data, i.e. whether it is a `<<SessionStart>> Message` or a `<<ConfigDataRequest>> Message`. In CASL, this can be specified by a free type construct

```
free type D_SI_Init ::= sort D_SI_Init_SessionStart
                      | sort D_SI_Init_ConfigDataRequest
                      | sort D_SI_Init_ConfigDataResponse
                      | ...
```

where each alternative corresponds to a message type occurring in the activity diagram.

But if on the component description level the concrete value of a message triggers a specific behaviour, it is necessary to specify the data types up to representation. Fig. 4 shows the different messages which the `Acquirer` might send to the `Terminal` in order to make requests on its configuration. These messages can be modelled by a CASL free type, and we can finally make concrete which data are involved in a `D_SI_Init_ConfigDataRequest`:

```
free type ConfDataObjRequest ::=
  ACD                    %% Acquirer Config Data
| AISD                   %% Acquirer Init Srv Data
| CPTD (ApplicationID)   %% Card Profile Table Data
| CAD  (RegisteredApplicationProviderID) %% Certification Auth Data
| TACD (ApplicationID)   %% Terminal Application Config Data
...
free type D_SI_Init_ConfigDataRequest ::=
 configDataRequest(ac:AcquirerID;term:TerminalID;conf:ConfDataObjRequest)
```

Up to now data modelling involved only sort declaration, sub-sorting and several forms of disjoint union via the free types construct. But on the component description level, also operations on data and axioms describing them come into play. We give a simple example, again from the context of the Terminal's initialisation. The ep2 documentation states here: *If the configuration download is started by the service man or the 'Use Case: Initiate Terminal Setup', the Terminal sets the* `<Config Download Mode>` *to '1' indicating 'Forced download' otherwise to '0' for 'Download check'.* We model this case distinction by a function `sessionStart` which is specified by the following axioms:

```
axioms sessionStart(serviceMan) = forcedDownload;
       sessionStart(initialTerminalSetup) = forcedDownload;
       sessionStart(others) = downloadCheck
```

Note that like in the modelling of the reactive behaviour, the different levels of data abstractions are clearly connected by refinement relations.

5 Results

Our overall experience of specifying ep2 in CSP-CASL is that while it's easy to formalize high level descriptions (e.g. the system architecture) from semi-formal descriptions (e.g. UML-like diagrams), writing specifications at the concrete level is more involved. At the more concrete levels, one has to deal with more unresolved and unclear descriptions (mostly presented as text), and decide which information must be formalized and what details should be ignored as they belong to other components or to another abstraction level. Having overcome these obstacles, the CSP-CASL formalization is again fairly straightforward. In this sense, our CSP-CASL specifications clearly mirror the ep2 document structure and specify at the different abstraction levels present therein.

As for CSP-CASL's expressivity, both the data types and the reactive behaviour present in ep2 can be adequately formalized. In modelling the data

types, CASL's subsorting feature proved particularly helpful. In modeling the reactive behaviour, CSP's distinction between internal and external choice was similarly important.

5.1 Resolution of shortcomings

Formalising ep2 in CSP-CASL leads to the partial or complete resolution of the problems outlined in section 2.3.

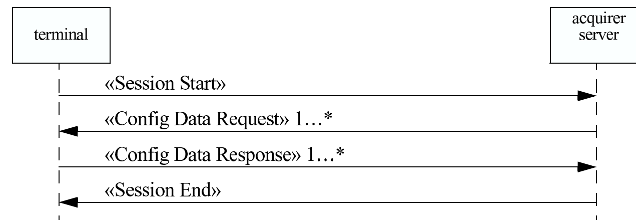

Fig. 7. Sequence diagram for requesting configuration data, as shown in [2].

One reason for this is that we are describing the system within *one* framework. In the *data modelling* for instance the possible values of the message type `<<Config Data Request>>` are described independently in various ep2 documents, where such different formats as text and tables are involved. Here, one of these texts mentions values `LAID` and `LRID` – see the excerpt in Section 2.2 – which do not appear anywhere else. CSP-CASL enables us to specify the corresponding data type only once and – via CSP-CASL's library mechanism – use it then in different contexts. If only data types are concerned, the CASL tool set CATS offers the possibility of static checks for inconsistencies. Looking on the *reactive side*, a comparison of the diagrams Fig. 3 and Fig. 7 shows that they specify the order of Requests and Responses differently: Fig. 3 requires that after one Request exactly one Response has to follow. In spite of this, Fig. 7 suggests that several Requests and several Responses can be 'bundled', and that a session might include different numbers of Requests and Responses. This inconsistency is clearly due to the change between the two formalisms involved (and maybe a weakness of the latter). In CSP-CASL, we can easily specify both variants; in our ep2 formalization we decided to follow Fig. 3.

Another aspect is that the specification language CSP-CASL itself guides us during the formalization process. CSP for instance is famous for its clarity concerning different forms of non-determinism. Thus, in modelling a diagram such as Fig. 3, it is natural to ask if the decision between the different branches is an internal or an external one. In this example, it is the Acquirer who takes the decision. Studying the documentation of the Acquirer further it turns out that for the purpose of the ep2 system it is unnecessary to model the database which is checked for 'Additional Configuration Data' in order to trigger the decision. This leads finally to a simple stateless process as a model for this part

of the Acquirer. Interestingly enough, in the description of different parts of ep2 the 'decision points' depicted as diamond with outgoing arcs in this kind of diagram need different formalizations in CSP-CASL: as internal non-determinism, as external non-determinism, and sometimes it is even the case that there is no decision to make. Concerning data, the loose semantics of CASL allows us to postpone design decisions until they are actually required. As seen in Section 4.2 on the different levels of data abstraction, sub-sorting is a powerful mechanism in decomposing complex data type into subtypes of manageable size.

The formalization helps also to design a certain system aspect only once, with the consequence of avoiding possible source of inconsistencies. For data types, this has been illustrated above with the message type `<<Config Data Request>>`. Concerning reactive behaviour, writing a CSP-CASL specification often helped to avoid over-specification. For instance, in the ep2 documentation the Terminal's responses to a request from the Acquirer are described at least twice: in the Terminal documents and in the Acquirer documents. In the world of CSP processes this is unnecessary: after sending a request to the Terminal, the Acquirer process wants only to receive a message on the channel which is connected with the Terminal. Only in the formalization of the Terminal is it necessary to state which specific response has to be sent, and as we have seen in Section 4.1, this is only necessary at a quite concrete level of abstraction.

5.2 Access to formal proofs

One of the benefits of specifying ep2 formally is that it makes it possible to establish properties by formal proofs on the CSP-CASL specifications describing the system. First experiments in this direction include proofs of refinement relations, deadlock analysis and consistency checks of the data types.

For instance, with the newly developed CSP-Prover [12] we were able to prove that

- our CSP-CASL specification corresponding to the activity diagrams 'Get Initialisation Data' — see Fig. 3 for the Acquirer's side of the protocol — is deadlock-free, and
- that — concerning the reactive part of the CSP-CASL specification of the activity diagrams 'Get Initialisation Data' — the specification on the concrete component description level refines[4] the abstract component description level.

Concerning data types, we used the CASL consistency checker [14] in order to prove the consistency of data types on the component description level. Here, we concentrated on the simple case of data types corresponding to ep2 messages as e.g. the `ConfigDataRequest`. At first glance this seems to be trivial, as on the CASL side these data types involve just a free datatype construct. But as the components of such a free type refer to other specifications, the question of consistency becomes a more involved problem as checking for non-interference between several separate specifications is required.

[4] Here, we use CSP's notion of stable-failure refinement.

6 Discussion and future work

We have shown how to specify a non-trivial system in the formal specification language CSP-CASL. Since ep2 is a prototypical example, the obtained results also hold for other systems such as web services, communicating financial agents, etc. Reconsidering our original aims, the specification language turned out to be well-suited to "translate" informal and natural language constructs, and rich enough to cover most important aspects of this particular system. Furthermore, it turned out that it is mostly possible, but not always advisable to follow the original document structure in the formalization. Considering scalability, we found that it is neither much harder nor much simpler to write a formal than an informal specification. In fact, we think that both styles have their own benefits; ideally the formal text should accompany informal descriptions in a 'literate specification'.

Related work The specification and implementation of banking software belongs to the most widely used exercises in computer science education. For example, in [4] the implementation of automated teller machine (ATM) software from an object oriented analysis and design is described. This graduate-level tutorial comprises a nice example of current best practice in software engineering, from the informal requirements specification up to an executable applet which can be used by students for testing purposes.

Similarly, many efforts have been invested in the verification of basic principles of the communication protocols which are employed in banking software. For example, in [8], some aspects of the *Millicent micropayment protocol* are modelled in an abstract protocol notation which is close to CSP, and security aspects are verified from this. As another example, in [18] authentication issues in the *Secure Electronic Transaction* (SET) protocol of Visa/Mastercard are verified by model checking a multi-agent logic of belief and time.

Not much work, however, has been mentioned in the formal specification and verification of real banking software and standards such as EMV or ep2. As an early example, in [26], the UNITY-method is used to refine a high-level specification of an electronic funds transfer system into one that could in principle be turned into an executable program. A more recent example of a formal specification of an actual banking standard is reported in [27], where the *Mondex electronic purse system* was proven correct with respect to its CSP and Z specification and was certified according to UK ITSEC Level 6. In [17], the *Internet Open Trading Protocol* (IOTP) is specified with colored Petri nets from an Request for Comments (RFC) by the Internet Engineering Task Force (IETF). In [6], it is argued that an interdisciplinary approach is necessary in this field, where experts from business administration, computer science and electrical engineering specify different views of a system. As example, a real internet based CD retail store system is specified in an integrated system model.

Future work Our next steps on formalizing ep2 will be to complete the modelling as far as possible. In particular, up to now we have formalized only a significant part of the whole specification, where the main omissions are the low-level XML

communication between actors and the security layer. In fact, the security part of ep2 heavily relies on common sense and external documents; in order to be able to prove security properties we will have to add certain assumptions about the underlying cryptographic methods. Other proofs on the formal model which we already started include refinement relations and deadlock analysis with CSP-Prover [12], as well as consistency of the data types [14]. Livelock analysis is to follow.

Finally, we want to use the model to automatically generate test cases for the different components of the ep2 system. It is an interesting research topic to define criteria which measure both data and control coverage of such test suites.

Acknowledgements The authors would like to thank Yoshinao Isobe (AIST) for help with refinement proofs and deadlock analysis of our specifications with CSP-Prover, Erwin R. Catesbeiana Jr (USW) for his advice on the modelling approach, and Christoph Schmitz and Martin Osley (Zühlke Engineering AG) for the good cooperation on ep2.

References

1. FM'99 exhibition: Competition 'Cash-Point Service', 1999. Based on the "9th Problem" stated at the Workshop on "The Analysis of Concurrent Systems", Cambridge, September 1983, LNCS 207.
2. *eft/pos 2000 Specification, version 1.0.1.* EP2 Consortium, 2002.
3. M. Bidoit and P. D. Mosses. CASL *User Manual.* LNCS 2900. Springer, 2004.
4. R. C. Bjork. Course notes 'Object-Oriented Software Development'. Department of Mathematics and Computer Science, Gordon College, Fall 2004. http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample.
5. B. T. Denvir, W. T. Harwood, M. I. Jackson, and M. J. Wray, editors. *The Analysis of Concurrent Systems*, LNCS 207. Springer, 1985.
6. A. Franz, P. Sties, and S. Vogel. Formal specification of e-commerce applications – an interdisciplinary approach. In K. Altinkemer and K. Chari, editors, *Proceedings of the Sixth INFORMS Conference on Information Systems and Technology.* ForSoft Publications, TU Munich, 2001.
7. M. Frappier and H. Habrias, editors. *Software Specification Methods.* Springer, 2001.
8. M. G. Gouda and A. X. Liu. Formal specification and verification of a micropayment protocol. In *Proceedings of the 13th IEEE International Conference on Computer Communications and networks, Chicago (Oct 2004).* IEEE Press. To appear.
9. J. F. Grote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing. Springer, 1995.
10. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
11. ISO 8807. Lotos — a formal description technique based on the temporal ordering of observational behaviour, 1989.
12. Y. Isobe and M. Roggenbach. Webpage on Csp-Prover. http://staff.aist.go.jp/y-isobe/Csp-Prover/Csp-Prover.html.
13. JTCI/CS7/WG14. The E-LOTOS final draft international standard, 2001.

14. C. Lüth, M. Roggenbach, and L. Schröder. CCC —the CASL Consistency Checker. In J. L. Fiadeoiro, P. D. Mosses, and F. Orejas, editors, *Recent Trends in Algebraic Development Techniques, 17th International Workshop, WADT 2004, Barcelona, Spain, March 27-30, 2004, Revised Selected Papers*, LNCS. Springer, to appear.
15. T. Mossakowski, M. Roggenbach, and L. Schröder. CoCASL at work — Modelling Process Algebra. In *Coalgebraic Methods in Computer Science*, volume 82 of *Electronic Notes Theoretical Computer Science*, 2003.
16. P. D. Mosses, editor. CASL *Reference Manual.* LNCS 2960. Springer, 2004.
17. C. Ouyang, L. M. Kristensen, and J. Billington. A formal and executable specification of the internet open trading protocol. In K. Bauknecht, A. M. Tjoa, and G. Quirchmayr, editors, *Proceedings of 3rd International Conference on E-commerce and Web Technology*, LNCS 2455, pages 377–387. Springer, 2002.
18. M. Panti, L. Spalazzi, and S. Tacconi. Verification of security properties in electronic payment protocols. In *Workshop on Issues in the Theory of Security (WITS '02); Co-located with IEEE POPL, Portland (Jan. 2002)*, 2002.
19. G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL — a CASL extension for dynamic Reactive Systems — Summary. Technical Report DISI-TR-99-34, Università di Genova, 2000.
20. G. Reggio and L. Repetto. CASL-CHART: a combination of statecharts and of the algebraic specification language CASL. In *Algebraic Methodology and Software Technology*, volume 1816 of *LNCS*, pages 243–257. Springer, 2000,.
21. H. Reichel, T. Mossakowski, M. Roggenbach, and L. Schröder. Algebraic-coalgebraic specification in CoCASL. In *Recent Developments in Algebraic Development Techniques, 16th International Workshop (WADT 02)*, LNCS 2755. Springer, 2003.
22. M. Roggenbach. CSP-CASL – A new integration of process algebra and algebraic specification. In G. S. F. Spoto and A. Nijholt, editors, *AMiLP 2003*, pages 229–243. University of Twente, 2003.
23. A. Roscoe. *The theory and practice of concurrency.* Prentice Hall, 1998.
24. G. Salaün, M. Allemand, and C. Attiogbé. A formalism combining CCS and CASL. Technical Report 00.14, University of Nantes, 2001.
25. G. Salaün, M. Allemand, and C. Attiogbé. Specification of an access control system with a formalism combining CCS and CASL. In *Parallel and Distributed Processing*, pages 211–219. IEEE, 2002.
26. M. G. Staskauskas. The formal specification and design of a distributed electronic funds transfer system. *IEEE Transactions on Computers*, 37, 1988.
27. S. Stepney, D. Cooper, and J. Woodcock. An Electronic Purse: Specification, Refinement, and Proof. Technical Monograph PRG-126, Oxford University Computing Laboratory, 2000.