Modelling Dynamic Data Structures with the B Method

Frédéric Badeau, Vincent Lacroix, Vincent Monfort, Laurent Voisin, and Christophe Métayer*

Systerel, Aix-en-Provence, France
{first.last}@systerel.fr,
http://www.systerel.fr

The software B method has so far been mainly used in the industrial world to develop safety critical software with very basic memory management limited to arrays of fixed size defined at compilation time.

We present here an alternative approach for modelling software based on a more classic memory management with dynamically allocated complex data structures accessed through pointers.

1 Context

Critical supervision systems are exposed to an increasing number of security threats, which have deep consequences in domains such as energy, transport, and defense. It is crucial to build components for the industry 4.0 and Industrial Internet of Things that can better resist such threats.

OPC-UA is a standard [6] for data exchange in industrial communications. It provides safe and secure means to connect supervision systems (SCADA) with programmable logic controller (PLC), actuators, and sensors.

INGOPCS is a research project that aims at developing an open source OPC-UA toolkit, named S2OPC[1]. This toolkit can be used for both server and client software. It has been designed to comply with both the EAL4 security level (Common Criteria [4]) and the SIL2 safety level (ISO/IEC 61508 [5]).

The toolkit is developed in C99. In order to ensure high assurance in the development, formal methods have been applied: B modelling of OPC-UA services, which is translated to C code; formal analysis of low-level manual C code using Frama-C and TrustInSoft Analyzer.

Systerel has strong experience in software B modelling for embedded railway applications. The toolkit however, does not address the same constraints:

- railway applications are cyclic (their entry point is a B operation called at each software cycle, that reads input messages, computes internal data and writes output messages), whereas the toolkit is triggered by events, either from the network or the application level,
- railway applications use static memory management, whereas the toolkit requires dynamic management, due to the wide diversity of exchanged objects.

* This work has been supported by an FUI 19 grant of the French government.

[1] See https://gitlab.com/systerel/S2OPC.

This paper explains how Systerel adapted its methods and design patterns to build a B model of the OPC-UA services.

2 Modelling Dynamic Data

Historically, the software B method [1] has been used in an operational context for the development of safety critical railway software [3, 2]. To meet these needs, the B language has evolved, B translators into classic programming languages, such as the C language, have been developed and methodological principles for software B development have been defined. This package B language / translator / software B methodology has reached a good level of maturity for the development of this type of software and has changed little since the early 2000s.

Although this approach is satisfactory, it is limited to the development of cyclic software where data is entirely managed within the model. All variables are statically allocated and contain simple data structures.

These features are not compatible with the development of software implementing the OPC-UA protocol that heavily uses record-based data structures, pointers to these data and dynamic allocation. We could try to get around these memory management issues by interfacing the C data structures with B arrays. However this approach is not satisfactory because it would imply a lot of data copying in between the C program and the B model.

We will see how to solve all these problems without changing either the B language or the B to C translators, thus bringing methodological solutions.

Classic B data structures. Implementable data structures supported by the software B language and the C translators are 1 or 2-dimensional scalar arrays and scalar types, where scalar types are the `int` type, the Boolean type, enumerated types and carrier sets. Attempts to introduce record types into the B-Language have raised major issues that made them impractical. For instance, modifying several fields of the same record in a row leads to very large and hard proof obligations. Therefore, with the classic B methodology, single records are usually replaced by several scalars and arrays of records are replaced by several arrays, one for each field of the record.

Carrier sets. One of the great advances of the current software B methodology has been to successfully use carrier sets to provide strong typing at the B level, while at the C level everything is integer. If we were to use integer subtypes instead of carrier sets, we could misuse a variable for another one belonging to a different subset, even at proof level, since all variables would be regular integers. For example, in a rail system, trains, signals or points may be modeled by carrier sets. Thanks to the strong typing, one is sure not to confuse a signal and a point in the B formulas.

Pointer management. To handle pointers, the key idea is to consider that a carrier set may represent a pointer type. For instance, to represent a pointer to a record of two Cartesian coordinates x and y, we define in a basic machine a

pointer type t_pos_i denoting all possible pointer values, a subset t_pos denoting pointers valid at some point in time and constant c_pos_undef denoting the NULL pointer value. The fields of the pointed record are modelled by partial functions. In addition, we define operations for memory management (allocation and de-allocation) and access to field values. The basic machine is implemented straightforwardly in C with the usual semantics.

SETS
  t_pos_i
ABSTRACT_CONSTANTS
  t_pos,
  c_pos_undef
PROPERTIES
  $\mathtt{t\_pos} \subseteq \mathtt{t\_pos\_i} \wedge$
  $\mathtt{c\_pos\_undef} \in \mathtt{t\_pos\_i} \wedge$
  $\mathtt{c\_pos\_undef} \notin \mathtt{t\_pos}$

VARIABLES
  f_pos_x,
  f_pos_y
INVARIANT
  $\mathtt{f\_pos\_x} \in \mathtt{t\_pos} \pfun \mathbb{Z} \wedge$
  $\mathtt{f\_pos\_y} \in \mathtt{t\_pos} \pfun \mathbb{Z} \wedge$
  $\mathrm{dom}(\mathtt{f\_pos\_x}) = \mathrm{dom}(\mathtt{f\_pos\_y})$

INITIALISATION
  $\mathtt{f\_pos\_x}, \mathtt{f\_pos\_y} := \varnothing, \varnothing$
OPERATIONS
  $\mathtt{p} \leftarrow$ pos_alloc $\mathrel{\widehat{=}}$
    CHOICE p := c_pos_undef OR
      ANY np, nx, ny
      WHERE $\mathtt{np} \in \mathtt{t\_pos} - \mathrm{dom}(\mathtt{f\_pos\_x}) \wedge \mathtt{nx} \in \mathbb{Z} \wedge \mathtt{ny} \in \mathbb{Z}$
      THEN $\mathtt{f\_pos\_x}(np) := \mathtt{nx} \parallel \mathtt{f\_pos\_y}(np) := \mathtt{ny} \parallel \mathtt{p} := \mathtt{np}$ END
    END
  pos_free(p) $\mathrel{\widehat{=}}$
    PRE $\mathtt{p} \in \mathrm{dom}(\mathtt{f\_pos\_x})$
    THEN $\mathtt{f\_pos\_x} := \{\mathtt{p}\} \ndres \mathtt{f\_pos\_x} \parallel \mathtt{f\_pos\_y} := \{\mathtt{p}\} \ndres \mathtt{f\_pos\_y}$ END
  $\mathtt{x} \leftarrow$ get_pos_x(p) $\mathrel{\widehat{=}}$ PRE $\mathtt{p} \in \mathrm{dom}(\mathtt{f\_pos\_x})$ THEN $x := \mathtt{f\_pos\_x}(p)$ END
  set_pos_x(p, x) $\mathrel{\widehat{=}}$ PRE $\mathtt{p} \in \mathrm{dom}(\mathtt{f\_pos\_x})$ THEN $\mathtt{f\_pos\_x}(p) := \mathtt{x}$ END

As pointers are viewed as scalars at the B level, nothing prevents us from defining more complex structures where a field value is itself a pointer to another record. We even can use arrays of pointers.

3 Use Case: OPC-UA Message manipulation

The decoding of incoming OPC-UA messages is an interesting application of this approach. Messages arrive on a network socket as a stream of bytes. These bytes are stored in a byte buffer (whose size is dynamic and unknown at compile time) by low-level C code. They are then interpreted by the B model to create a structured message, that is a record containing itself other records and arrays. Again, this message structure is dynamic and depends on the contents of the buffer, so it cannot be pre-allocated.

A pointer to the byte buffer is passed as input to a B model entry point. This buffer is already allocated and passed as a valid pointer. The buffer state variable is initialized as not read yet. The buffer can then be read in a predefined order that is enforced by buffer states used as preconditions for decoding operations (message type, message header and message). These decoding operations allocate message structures. In case of failure the buffer becomes unreadable (undefined state). When reading is finished, the buffer is invalidated by setting its state to undefined to prevent misuse.

The input message is allocated by the decoding operation. It is thereon used as a valid message pointer. Then, messsage fields can be accessed as stated in Sect. 2. Once the message content has been consumed, it is deallocated and the pointer is set as invalid.

This example shows how the B model encapsulates the low-level byte buffer and message structures. It controls the structure manipulation and the possible field accesses. The memory management is also driven by the B model, even though the object lifecycle is partial in it (e.g., an output pointer remains valid out of the model but is considered invalid in it). Basic machines are only used for low-level and basic operations (actual decoding, structure accesses).

4 Conclusion

The new approach that we have described allows us to incorporate explicitly dynamic memory management in our B model, including allocation, deallocation and the validity of pointers. Since these properties are represented in the model, they can be used in B invariants and operations to provide guarantees when manipulating data referenced by pointers. As a consequence, operations can define pre-conditions to manipulate only valid pointers and have read/write access to the underlying data structures. Moreover the input and output data flow can also be treated in the model even if part of the memory management is done upstream or downstream of the model.

Another important aspect is that the data structures and their associated low-level services (implemented directly in C) can be encapsulated in the B model to add high-level properties on the services which are guaranteed by a formal proof.

Thanks to the methodological advances presented, we successfully developed a B model integrated in a piece of C software manipulating dynamically allocated data and pointers, notably on structures.

There are still a few issues that we did not address yet. Firstly, all our data structures contain a fixed number of elements whose size is also statically fixed. One could extend the approach to dynamically sized fields and variant records (e.g., C unions). Secondly, for recursive data structures (e.g., linked lists, trees) one may also want to express some (inductive) property on the data structure as a whole, while our approach is currently limited to each record (e.g., cell) independently of the others.

References

1. J. Abrial. *The B-book - Assigning programs to meanings.* Cambridge University Press, 2005.
2. F. Badeau and A. Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In *ZB 2005*, volume 3455 of *LNCS*, pages 334–354. Springer, 2005.
3. P. Behm, P. Benoit, A. Faivre, and J. Meynadier. Météor: A successful application of B in a large project. In *FM'99*, volume 1708 of *LNCS*, pages 369–387. Springer, 1999.
4. Common Criteria. Common criteria for information technology security evaluation. CCMB 2017-04-001, Common Criteria Portal, 2017.
5. IEC. Functional safety of electrical/electronic/programmable electronic safety-related systems. IEC 61508:2010, International Electrotechnical Commission, Geneva, Switzerland, 2016.
6. IEC. OPC unified architecture. IEC TR 62541:2016, International Electrotechnical Commission, Geneva, Switzerland, 2016.