SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 02-36

Model Checking Object-Z using ASM

Kirsten Winter and Roger Duke

Version 1, October 2002

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

Published as:

Kirsten Winter, Roger Duke: Model Checking Object-Z using ASM In K. Sere and M. Butler (eds.), *3rd International Conference on Integrated Formal Methods (IFM 2002)* , volume 2335 of Lecture Notes in Computer Science, pages 165-184. Springer-Verlag, 2002.

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via `http://svrc.it.uq.edu.au`

Model Checking Object-Z using ASM

Kirsten Winter and Roger Duke

Software Verification Research Centre
School of Information Technology and Electrical Engineering
University of Queensland
phone: +61 7 3365 1638 fax: +61 7 3365 1533
`kirsten@svrc.uq.edu.au` `rduke@itee.uq.edu.au`

Abstract. A major problem with creating tools for Object-Z is that its high-level abstractions are difficult to deal with directly. Integrating Object-Z with a more concrete notation is a sound strategy. With this in mind, in this paper we introduce an approach to model-checking Object-Z specifications based on first integrating Object-Z with the Abstract State Machine (ASM) notation to get the notation OZ-ASM. We show that this notation can be readily translated into the specification language ASM-SL, a language that can be automatically translated into the language of the temporal logic model checker SMV.

Keywords: Object-Z, Abstract State Machines, language transformation, model checking, automated tool support.

1 Introduction

High level declarative specification languages such as Z [20] and Object-Z [3, 19] are ideally suited for capturing state-based system properties. A major obstacle to the wider adoption of such languages, however, is the lack of tool support. If the full potential of such languages is to be realised, they need to be supplemented with software tools to assist with the more taxing mathematical aspects such as the detection of specification errors and the verification of system properties.

In this paper we introduce an approach to model-checking Object-Z specifications based on first integrating Object-Z with the Abstract State Machine (ASM) notation [8] and then, through an automated process, translating this integrated notation (which we call OZ-ASM) into the temporal logic model checker SMV [16].

We chose this integrated approach for several reasons. First, when faced with the task of model checking Object-Z specifications it seemed wise to see if we could make maximum use of existing model-checker technology rather than reinventing the wheel.

Next, the problem with Object-Z by itself is that, based as it is on logical predicates and set theory, its high-level abstractions are difficult to deal with directly. Most model checking tools do not provide a language that is expressive enough to represent the operators and data structures that can be used in Object-Z. An automatic transformation of the full language of Object-Z into a simple model checker language would be hard to achieve. However, if we re-use an existing interface from ASM to the SMV language [1] we only need to provide a transformation from Object-Z into ASM in order to automatically generate SMV code from Object-Z models.

Third, for an algorithmic transformation from Object-Z into ASM, we have to ensure that operation predicates are given in a canonical form: the transition relation that relates pre- and post-states must be given in such a way that primed variables (denoting post-state values) depend only upon unprimed variables (denoting pre-state values). To provide this canonicity, we integrate Object-Z with ASM syntax. The resulting integrated language is easy to read and understand for anybody who is familiar with Object-Z.

Our work has similarities to that of Valentine [21] who introduces a restricted version of Z, called Z--, which has a similar canonicity of predicates. However, Valentine aims to provide a computational subset of Z, and the language restrictions required to do this are stronger than those required for model checking. In contrast to Valentine's approach, we use the concept of language integration to give a simple definition for the restricted scope of the language.

Based on Jackson's work on the automatic analysis of a subset of Z [12], Jackson et. al. have developed a new language, Alloy [13], which can be automatically analysed through SAT solvers. The language Alloy, created to avoid working with an 'ad hoc restriction' of Z, satisfies the necessary conditions for being translated into boolean formulae (a format that SAT solvers can deal with). In accordance with the SAT solver approach, Alloy is a declarative language. In contrast, our language focuses on operational models as required for model checkers.

The use of SMV to model check Z specifications has been suggested by Jacky and Patrick [14]. Their approach is to translate the Z specification directly into SMV, a process that involves considerable simplification of the specification. In contrast, our approach is to first integrate Object-Z with ASM. The OZ-ASM notation that results can be readily translated into the specification language ASM-SL and this can be automatically translated into SMV. Furthermore, ASM-SL is an ideal starting point for the application of other tools (although in this paper we will consider only SMV model checking).

Other researchers have approached the problem of model-checking Object-Z using tools based on process algebras such as FDR [6] or SPIN [11]. For example, Fischer and Wehrheim [5] integrate Object-Z with CSP and apply the model-checker FDR. Kassel and Smith [15] investigate an alternative approach for plain Object-Z but also exploit the CSP semantics for their mapping into CSP_M (the language of FDR). Although CSP_M is sufficiently expressible to cover many language constructs provided for Object-Z predicates, neither object referencing nor operation operators are supported. Moreover, the compilation effort within the FDR tool is enormous even for small examples. Similarly, Zave [25] has looked at model-checking Z by combining Z with Promela [10] (the language of SPIN).

However, although Promela is designed for model checking, it is not suited for representing system state or for manipulating global data.

Yet another approach to the checking of Z is to animate the specification. Essentially, Valentine's work mentioned previously [21] follows this strategy. Another computational approach is given by Grieskamp [7] who constructs a computational model for Z that allows the combination of the functional and logical aspects of Z.

In this paper we begin (in Section 2) by introducing the OZ-ASM specification notation integrating Object-Z and ASM. It is specifications written in OZ-ASM that we will be model checking. We emphasise, however, that we do not see OZ-ASM as yet another independent formal specification notation. Our underlying wish is to model-check Object-Z specifications, and ideally a first step in fulfilling such a wish would be to convert an Object-Z specification into the OZ-ASM notation. In practice, however, although such a conversion is in most cases straightforward, as it involves (among other things) various non-deterministic choices about the cardinality of underlying set structures, it would be difficult if not impossible to perform such a conversion automatically. For this reason, in this paper we take the OZ-ASM notation as our starting point.

Specifications written in ASM-SL can be automatically translated to SMV for model checking [1]. In our case the starting notation is OZ-ASM rather than ASM-SL, but the process of translation to SMV is essentially the same. In Section 3 we outline the principle issues behind the conversion from OZ-ASM into ASM-SL. Once this conversion is completed, the translation into SMV is automatic. We believe that it would be straightforward to automate the conversion from OZ-ASM into ASM-SL; we leave that for future work.

In Section 4 we illustrate how to apply the given transformation rules to convert a complex OZ-ASM operation into ASM-SL code. We discuss future directions and conclude the paper in Section 5.

2 The Integrated Language OZ-ASM

Object-Z [3, 19] is a high-level specification language based on Z [20]. It supports object-orientation through the concept of class, inheritance between classes and object referencing. Object-Z models of complex systems can be nicely structured into a set of smaller sub-systems or components. Each component is modelled as a class that contains its local definitions: a state, an initialisation and operations. Object instances are accessed through referencing.

Most of these concepts are not supported by a simple model checker language, which is in most cases designed to describe hardware circuits, and a transformation from Object-Z into a model checker language is difficult to build. In order to ease this task we propose to make use of an existing transformation, the interface from the high-level language ASM to the SMV language ([1]). In doing so, it only remains to develop a transformation from Object-Z into ASM, a task that is easier to achieve since most Object-Z concepts can be simulated with ASM.

ASM [8] is a formal specification language that models state transition systems at a high level of abstraction. However, object orientation with its benefits for modular design is not facilitated. The concept of classes and, especially, the reuse of classes via inheritance and object instantiation is not supported.

The transition relation of an ASM model is given through a set of transition rules for which ASM provides a minimal set of rule constructors. These rule constructors are sufficient to model any kind of state machine or (sequential) algorithm [9]. The core concept of an ASM transition rule involves *locations* which become *updated* in a transition step, i.e. $loc := val\_term$. Similar to guarded command languages, updates can be conditional in the sense that the update is fired only if the corresponding guard is satisfied in the current state, i.e. `if` *guard* `then` $loc := val\_term$. Clearly, a location resembles a state variable and the concept of guarded updates coincides with the notion of a transition relation in the model checker languages we are targeting. Note, however, that the two terms, *guard* and *val_term*, depend only on the evaluation of terms in the current state; no next-state value can occur.

Whilst the structuring mechanisms of Object-Z are very close to concepts of object-oriented programming languages, the modelling paradigm for operations is essentially the same as in Z. That is, operations are modelled in a declarative way rather than operationally. The model represents *what* are the requirements of a system. The set of acceptable behaviours is thus specified implicitly by means of predicates that define a relation between pre- and post-states. However, in many cases, an operational modelling approach is adopted in which operations are seen as a transition in a state machine that describe an actual step, or sequence of steps, between pre- and post-states. The specification can present some information on *how* the requirements are met by the system. The user of Object-Z is basically free to choose a modelling style and the degree of abstraction of the model. No restrictions are given by the language.

If we want to provide an interface between Object-Z and ASM, we have to transform the definition of operations (i.e. the relation between pre- and post-states) into the canonical form of a transition relation that is supported by ASMs, i.e. each primed variable depends only on unprimed variables. For arbitrary Z or Object-Z predicates this transformation would be difficult (if not impossible) to perform automatically. Therefore, we restrict the appearance of predicates in an operation schema to a form that can be treated algorithmically. To do so, we integrate Object-Z with ASM.

For our integration of Object-Z and ASM, which we call OZ-ASM, we borrow the following ASM transition rule constructors[1]:

- `skip`
  The skip rule specifies an empty transition in which the state does not change.

[1] We do not allow the use of import or extend rules since these cannot be treated by the model checking approach ([1]).

– `loc := val_term`
  A simple update rule that specifies the assignment of a value to a location. `val_term` is a term whose value depends only on the current state.
– `if` *guard* `then` R_0 `else` R_2
  The if-then-else rule specifies a boolean expression *guard* (which may include also existential and universal quantification) and a rule that is to be fired in the case when the guard is satisfied, i.e. the *then-case*, as well as an optional rule for the case when the guard is not satisfied, i.e. the *else-case*.
– `block` R_1 R_2 ... R_n `endblock`
  The block rule allows the user to combine several rules into one. All rules $R_1, R_2, \ldots, R_n$ contained in the block rule are fired simultaneously.
– `do forall` v `in` A `[ with` G `]` $R(v)$ `enddo`
  The do-forall rule allows the user to parameterise a transition rule. The inner rule R is instantiated for all specified values v in set A and all rule instances are fired simultaneously. The optional boolean expression G allows the user to further restrict the possible values for v.
– `choose` v `in` A `[ with` G `]` $R(v)$ `endchoose`
  The choose rule allows the user to model non-determinism.

The listed ASM rule constructors replace ordinary predicates within operation schemas of OZ-ASM. Primed state variables do not occur anymore. Instead we use the variable assignment ':=' with a next-state variable as its left hand side. Thus, the use of predicates and their appearance in operation schemas is restricted in the intended way. The restriction, however, is only syntactical since any operation predicate (we could think of) can be 'simulated' by an ASM rule construct. Naturally, some expressions become more verbose but the semantics, adopted from Object-Z, will essentially be the same. The only significant semantic distinction between Object-Z and OZ-ASM is to do with the occurence of operations. In Object-Z an operation can occur if and only if it is applicable, i.e. its pre-conditions are satisfied and there exists a valid post-state. In OZ-ASM however, an operation can always occur; if it is not applicable (i.e. if the guard on the update rule is not true or if there is no valid next state) then the operation can skip, i.e. do nothing. Both non-occurrence and skip, however, have the same effect; they do not change the state of the systems.

As a consequence, operations that contain only a guard (i.e. a predicate over non-primed variables, e.g. *accessGranted* of class *Key* in the example on the following page) have the same effect if they are applicable or not. They do not change the state in both cases. These operations only become meaningful when combined with other operations (e.g. operation *insertKey* in class *KeySystem*). They restrict the applicability of the overall operation since their guard contributes to the overall guard (which is a conjunction of the guards of combined operations, see Section 3).

The benefit we gain from the syntactical adaption of ASM rules is a simpler language definition. Instead of listing a set of rules that define the predicates we can support, we provide (for Object-Z users) a new syntax of transition rules. These rule constructs will be easier to learn (and to keep in mind while modelling) than restrictions on predicates.

As an illustration of OZ-ASM we specify a simple key system (for the original Object-Z specification see [3] Chapter 3). This system consists of a set of keys and a set of rooms. Each key has access rights to (i.e. can unlock) a subset of the rooms, and these access rights can be modified, either by extending the access rights of a key to a room which the key cannot currently access, or by rescinding access rights to a room which the key can currently access. A key can attempt to access a room; if that room is locked and the key has access rights to the room, the room unlocks, otherwise nothing happens. An unlocked room can non-deterministically lock (in practice this would probably be realised by a timeout).

Key

$rooms : \mathbb{F}\ Room$

INIT
$rooms = \varnothing$

extendAccess
$\Delta(rooms)$
$rm? : Room$

if $rm? \notin rooms$
then $rooms := rooms \cup \{rm?\}$

rescindAccess
$\Delta(rooms)$
$rm? : Room$

if $rm? \in rooms$
then $rooms := rooms \setminus \{rm?\}$

accessGranted
$rm? : Room$

if $rm? \in rooms$
then skip

accessDenied
$rm? : Room$

if $rm? \notin rooms$
then skip

Room

$locked : \mathbb{B}$

INIT
$locked$

supplyId
$rm! : Room$

if $rm! = self$ **then skip**

unlock
$\Delta(locked)$

if $locked$
then $locked := false$

lock
$\Delta(locked)$

if $\neg locked$
then $locked := true$

KeySystem

$keys : \mathbb{F}\ Key$
$rooms : \mathbb{F}\ Room$

$\forall k : keys \bullet k.rooms \subseteq rooms$

Init
$\forall k : keys \bullet k.\textit{Init}$
$\forall r : rooms \bullet r.\textit{Init}$

$extendAccess \mathrel{\widehat{=}} [k? : keys] \bullet k?.extendAccess$
$rescindAccess \mathrel{\widehat{=}} [k? : keys] \bullet k?.rescindAccess$
$lock \mathrel{\widehat{=}} [\!]\ r : rooms \bullet r.lock$
$insertKey \mathrel{\widehat{=}} [r? : rooms;\ k? : keys] \bullet$
$\qquad r?.suppyId \parallel (k?.accessGranted \wedge r?.unlock$
$\qquad\qquad [\!]\ k?.accessDenied)$

Note that apart from predicates in operation schemas all the other constructs of Object-Z are not affected by our integration. State schema, state invariants, initialisation schema, and the definition of operations by means of operation operators can be used in OZ-ASM as in Object-Z. We claim that these constructs can be automatically mapped into a canonical representation that can be model checked. This mapping will be introduced and discussed in the next section.

3 Mapping OZ-ASM into ASM-SL

Castillo and Winter [1, 23] have shown that ASM models in general can be automatically transformed into a simple intermediate format for transition systems (namely, ASM-IL) which can in turn be readily mapped into languages like the SMV language ([16]). In particular, this transformation has been implemented for ASM-SL which is a specification language that provide a machine readable syntax for ASM.[2]

Our intention is to make use of this existing work by mapping OZ-ASM into ASM-SL. Because of the existing automatic transformation from ASM-SL into SMV, effectively we will have a transformation from OZ-ASM into SMV code. In the following we use the symbols $\Rightarrow$ and $\Downarrow$ to denote the mapping from OZ-ASM into ASM-SL expressions that provides the basis of the transformation algorithm to be implemented.

3.1 Types

Since the model checking approach that is utilised here is limited to systems with a finite state space, we have to assume that all types of an OZ-ASM system are finite. Moreover, any given type needs to be enumerated, i.e.

[2] ASM-SL is supported by the ASM-Workbench ([2]) and the transformation from ASM-SL into SMV code implements an interface from the ASM-WB to the SMV model checker.

$$[\mathit{GivenType}] \;\Rightarrow\; \texttt{datatype}\ \mathit{GivenType} == \{e_1, \ldots, e_n\} \tag{1}$$

where the number n of elements is explicitly defined. An approach for checking systems with given types is proposed in [24] but so as to be able to make use of an existing model checking tool we follow here the simpler approach introduced in [1].

3.2 Classes

As a general and readily implemented approach for simulating OZ-ASM by ASM-SL, we propose here a transformation that flattens the modular structure of OZ-ASM. In OZ-ASM, the concept of class provides a local name space for attributes and operations. These local name spaces are 'simulated' in an ASM-SL specification by identifiers that include the class name. Any types, constants or attributes that are local within a class are mapped into corresponding types or functions whose identifiers are extended by the class name. The use of functions allows us to deal easily with object referencing. For example, an attribute *attr* in a class A which is of type *AttrType* becomes a function $\mathit{A\_attr} : \mathit{AType} \rightarrow \mathit{AttrType}$. Referencing of the attribute *attr* of an object a is then realised by the function application $\mathit{A\_attr}(a)$. To achieve this we have to generate an ASM type for each class. Note that the number of instances of a class must be finite and fixed if we want to apply model checking.

Since ASM-SL is a strongly typed language we need to distinguish between types and sets. Types are used in the declaration of functions; sets, modelled as static functions, are used within terms. Due to this restricted use of declared types, we have to introduce for each OZ-ASM class both a type and a constant set, with both containing the same elements.

$$\begin{array}{|l}
\hline A \\
\ldots \\
\hline
\end{array}$$

$$\Downarrow \tag{2}$$

$$\begin{array}{l}
\texttt{datatype}\ \mathit{AType} == \{\mathit{obj}_1, \ldots, \mathit{obj}_n\} \\
\texttt{static function}\ A == \{\mathit{obj}_1, \ldots, \mathit{obj}_n\}
\end{array}$$

Constant attributes are mapped into static functions in ASM-SL. All attributes that are declared in the state schema are mapped into dynamic functions. Input attributes become external functions (which in ASM-SL correspond to the notion of input). Instead of the '?' decoration we add the extension '_in' to the name of the corresponding external function; similarly, the decoration '!' is replaced by the extension '_out'.

Suppose in the following that Type_2, Type_3, and Type_4 are already declared types and *AType* is given as above. Then the class A and its attributes as given below are mapped in the following way:

A

$Type_1 \ ::= \ t_1 \mid t_2 \mid \ldots \mid t_m$

$\mid attr_1 : Type_2$

$attr_2 : Type_3 \rightarrow Type_4$

$$\Downarrow \tag{3}$$

datatype $A\_Type_1 == \{A\_t_1, \ldots, A\_t_m\}$
static function $A\_attr_1 : AType \rightarrow Type_2$
dynamic function $A\_attr_2 : AType \rightarrow (Type_3 \rightarrow Type_4)$

Attributes of a set type are mapped into a characteristic function which maps each element that is a member of the set to *true* and each element that is not a member of the set to *false*.

A

$attr_3 : \mathbb{F}\ Type_2$

INIT
$\ldots$

$$\Downarrow \tag{4}$$

dynamic function $A\_attr_3 : AType \rightarrow (Type_2 \rightarrow BOOL)$
  initially $\{\ldots\}$

The initialisation of a dynamic function is derived from the INIT schema of the OZ-ASM model. We give an example:

Room

$locked : \mathbb{B}$

INIT
$locked$

Key

$rooms : \mathbb{P} Room$

INIT
$rooms = \varnothing$

extendAccess
$\Delta(rooms)$
$rm? : Room$

$\ldots$

$\Downarrow$ via (2), (3), (4)

```
datatype $RoomType == \{r_1, \ldots, r_4\}$
datatype $KeyType == \{k_1, \ldots, k_5\}$
static function $Room == \{r_1, \ldots, r_4\}$
static function $Key == \{k_1, \ldots, k_5\}$
dynamic function $Room\_locked : RoomType \to BOOL$
      initially MAP_TO_FUN $\{r \mapsto$ true $\mid r \in Room\}$
dynamic function $Key\_rooms : KeyType \to (RoomType \to BOOL)$
      initially MAP_TO_FUN $\{(k, r) \mapsto$ false $\mid$
                          Union $(\{\{(k, r) \mid k \in Key\} \mid r \in Room\})\}$
external function $Key\_rm\_in : KeyType \to RoomType$
```

3.3 Operations

When mapping OZ-ASM into ASM-SL we have to take into account that the underlying semantics of operation execution in OZ-ASM is different to the semantics of running an ASM program. In ASM all transitions fire simultaneously; in OZ-ASM one operation is selected for execution by angelic choice. This provides an asynchronous execution model.

As the target language SMV supports the notion of asynchronous processes, a semantic-preserving mapping from OZ-ASM into the SMV language can be given by the following scheme. Each OZ-ASM operation is transformed into a single-step ASM which we call the corresponding *operation*-ASM. An operation-ASM consists of one (possibly nested) transition rule that 'simulates' the operation.

Figure 1 illustrates the mapping from OZ-ASM operations into a set of operation-ASMs. Applying the results from [23], each operation-ASM can in turn be automatically transformed into an SMV process. In the corresponding SMV code all processes are tied together asynchronously by using the keyword `process` (see [16]). Semantically, in each state exactly one of the asynchronous processes is active and can proceed with one state change (or no state change if no progress is possible).

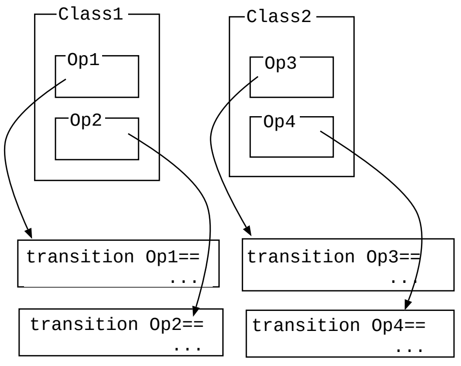

Fig. 1: The mapping schema

In the remainder of this section we introduce how OZ-ASM operations are mapped into operation-ASMs (in ASM-SL). We describe a mapping for operation schemas, the conjunction operator, the general choice operator, and the treatment of state invariants. The mapping for all the other operation operators, namely $\bullet$, $\|_!$, $\|$, and $\fatsemi$, can be derived from these basic operators (see [3, 19] for definitions).[3] However, in order to show the transformation on an example (see

[3] The sequential operator $\fatsemi$, however, should be restricted to the composition of two operations only.

Section 4), we extend this section by outlining the rules for promotion, scope enrichment, and parallel composition.

We assume in the following that the state space for any resulting operation-ASM has already been generated by making appropriate use of the global definitions described above.

Mapping Set Expressions Since in OZ-ASM operation predicates are expressed in the syntax of ASM transition rules, the mapping of these OZ-ASM transitions into ASM-SL is reduced to a mapping of expressions and locations that occur within the transitions. We introduce the function $\langle\langle . \rangle\rangle$ as a shorthand for the results of the transformation application, i.e. $\langle\langle A \rangle\rangle \equiv B$ holds if $A \Rightarrow B$, and give an (incomplete) set of rules for handling expressions. Assume that E, E_1, E_2, E_3 are sets where $E_1 \subset E$, $E_2 \subset E$, and $E_3 \subset E$. χ_{Ei} (for $i \in \{1, 2, 3\}$) denotes the characteristic function for the set E_i.

$$\langle\langle \texttt{if } g \texttt{ then } upds \rangle\rangle \equiv \texttt{if } \langle\langle g \rangle\rangle \texttt{ then } \langle\langle upds \rangle\rangle \tag{5}$$

$$\langle\langle e \in E \rangle\rangle \equiv \chi_E(e) = true \tag{6}$$

$$\langle\langle E_1 \subset E_2 \rangle\rangle \equiv \texttt{forall } e \texttt{ in } E : \tag{7}$$

$$\chi_{E1}(e) \texttt{ implies } \chi_{E2}(e)$$

$$\langle\langle E_1 := E_2 \rangle\rangle \equiv \tag{8}$$

```
block
  do forall $e$ in $E$
     with $(\chi_{E2}(e) = true)$
  $\chi_{E1}(e) := true$
  enddo
  do forall $e$ in $E$
     with $(\chi_{E2}(e) = false)$
  $\chi_{E1}(e) := false$
  enddo
endblock
```

$$\langle\langle E_1 := E_1 \cup \{e\} \rangle\rangle \equiv \chi_{E1}(e) := true \tag{9}$$

$$\langle\langle E_1 := E_1 \setminus \{e\} \rangle\rangle \equiv \chi_{E1}(e) := false \tag{10}$$

$$\langle\langle E_1 := E_2 \cap E_3 \rangle\rangle \equiv \tag{11}$$

```
block
  do forall $e$ in $E$
 with $(\chi_{E2}(e) = true$ and $\chi_{E3}(e) = true)$
  $\chi_{E1}(e) := true$
  enddo
  do forall $e$ in $E$
     with $not(\chi_{E2}(e) = true$ and $\chi_{E3}(e) = true)$
  $\chi_{E1}(e) := false$
  enddo
endblock
```

Although this set of rules is incomplete, it illustrates how set operations and expressions in OZ-ASM can be transformed into ASM-SL transition rules and expressions over characteristic functions. Similarly, rules for operators on other data types, such as sequences, can be defined.

Following the results in [23], all ASM rule constructs (apart from the extend and import rules) can be mapped into a set of simple rules of the form '`if` guard `then` updates'. For such a straightforward rule we can readily determine its guard, its locations and its updates. The guard is simply a boolean expression over variables, while locations are the variables that are changed by the rule (i.e. the left hand sides of updates). For updates we consider a set of assignment expressions of the form $loc := val$. We reference these entities in the following discussion with $guard(rule)$, $locs(rule)$ and $upds(rule)$ respectively.

Operation Schema Given the mapping for OZ-ASM transitions, we can now define rules for mapping OZ-ASM operations into operation-ASMs (in ASM-SL). We start with the rule for mapping operations that are defined simply in terms of an OZ-ASM operation schema. Due to the integration of ASM syntax into OZ, this rule is straightforward:

```
┌ A ──────────────────────
│ ┌ op ───────────────────
│ │ Δ(attributes)
│ ├──────
│ │ transition
│ └───────────────────────
└─────────────────────────
```

$$\Downarrow \tag{12}$$

$$\texttt{transition}\ A\_op == \langle\!\langle transition \rangle\!\rangle$$

Similarly, this rule applies to any operation definition, i.e. $op \mathrel{\widehat=} op_1 \Rightarrow$ $\texttt{transition}\ op == \langle\!\langle op_1 \rangle\!\rangle$.

Conjunction Operator In OZ-ASM the conjunction $op_1 \wedge op_2$ of two operations is applicable if the preconditions of both op_1 and op_2 are satisfied and for both operations there exists a valid post-state. The precondition is given by the conjunction of the guards of the mapped operations $\langle\!\langle op_1 \rangle\!\rangle$ and $\langle\!\langle op_2 \rangle\!\rangle$ together with any given state invariant that affects the operation.

The effect of the conjoined operation is basically given as the union of the updates $upds(\langle\!\langle op_1 \rangle\!\rangle)$ and $upds(\langle\!\langle op_2 \rangle\!\rangle)$, assuming that no conflicts occur. No conflict occurs if for any location that is addressed in both operations the different updates deterministically assign the same value to the location, or if the updates are non-deterministic but the intersection of both sets of possible update-values is not empty. No conflict occurs also in the mixed case, i.e. if the assigned value of a deterministic update falls into the range of a non-deterministic update.

In Object-Z, examples of the second situation arise quite frequently, e.g. $(n' \in \{1..10\}) \wedge (n' \in \{5..15\})$. The result of this conjunction is clearly $(n' \in \{5..10\})$, i.e. n' is a member of the intersection of both update-ranges. In OZ-ASM we model such non-deterministic updates by means of the choose-rule: `choose` v `in` $\{1..10\} \cap \{5..15\}\ n := v$ `endchoose`. The possible update-values of this rule are given as the range of v.

Let the function *upd_vals* provide for each rule R and each location *loc* that is addressed within R the set of possible update values that can be assigned to *loc*. For a deterministic update, $upd\_vals(R, loc)$ is a singleton that contains exactly the right hand side of the update, i.e. *val* for the update $loc := val$. (Note that *val* is the evaluation of the right hand side expression in the current state.) For a non-deterministic update we get more than one possible update value, so $upd\_vals(R, loc)$ is a set. For example, for the rule

```
transition $R$ ==
          choose $v$ in $\{1, \ldots, 10\}$
                $n := v + 1$
          endchoose
```

we get $upd\_vals(R, n) = \{1 + 1, \ldots, 10 + 1\} = \{2, \ldots, 11\}$.

In general, we can conjoin two operations that address the same location by selecting as the update value an element from the intersection of all the update-value sets for that location. All other updates that address locations that are not referenced in both operations can be fired simultaneously.

We denote the set of conflicting updates by means of a function *conflict_upds*:

$$\begin{aligned}
&(\texttt{if}\ g\ \texttt{then}\ loc := exp) \in conflict\_upds(\langle\!\langle op_1 \rangle\!\rangle, \langle\!\langle op_2 \rangle\!\rangle) \\
\Leftrightarrow\ & \ loc \in (locs(\langle\!\langle op_1 \rangle\!\rangle) \cap locs(\langle\!\langle op_2 \rangle\!\rangle)) \\
& \ \wedge\ (\texttt{if}\ g\ \texttt{then}\ loc := exp) \in (upds(\langle\!\langle op_1 \rangle\!\rangle) \cup upds(\langle\!\langle op_2 \rangle\!\rangle))
\end{aligned}$$

The set of non-conflicting updates of operations op_1 and op_2 is then determined by $(upds(\langle\!\langle op_1 \rangle\!\rangle) \cup upds(\langle\!\langle op_2 \rangle\!\rangle)) \setminus conflict\_upds(\langle\!\langle op_1 \rangle\!\rangle, \langle\!\langle op_2 \rangle\!\rangle)$.

Putting this all together we get the following rule for the conjunction of operation ASMs. (Note that sets of updates denote the simultaneous firing of these updates. The treatment of state invariants is discussed later in this section.)

$$op_1 \ \wedge \ op_2$$

$$\Downarrow \tag{13}$$

```
if $guard(\langle\!\langle op_1 \rangle\!\rangle)$ and $guard(\langle\!\langle op_2 \rangle\!\rangle)$ and $\langle\!\langle state\_invariants \rangle\!\rangle$
then
   if forall $loc$ in $(locs(\langle\!\langle op_1 \rangle\!\rangle)$ intersect $locs(\langle\!\langle op_2 \rangle\!\rangle))$
         $(upd\_vals(op_1, loc)$ intersect $upd\_vals(op_2, loc)) \neq$ emptyset
   then
      $(upds(\langle\!\langle op_1 \rangle\!\rangle) \cup upds(\langle\!\langle op_2 \rangle\!\rangle)) \setminus conflict\_upds(\langle\!\langle op_1 \rangle\!\rangle, \langle\!\langle op_2 \rangle\!\rangle)$
      do forall $loc$ in $(locs(\langle\!\langle op_1 \rangle\!\rangle)$ intersect $locs(\langle\!\langle op_2 \rangle\!\rangle))$
         choose $val$ in $(upd\_vals(op_1, loc)$ intersect $upd\_vals(op_2, loc))$
            $loc := val$
         endchoose
      enddo
   else skip
   endif
endif
```

Choice Operator The choice operator models an angelic choice between two operations. That is, depending on the applicability of the operations, either one or the other is executed. Only if both operations are not applicable in the current state is the choice operation not applicable.

We map a choice operation $op1 \;[]\; op2$ into an ASM that models the same behaviour. Generally, an operation-ASM is applicable if its guard is satisfied and the state invariant is satisfiable in the next state (see the discussion of state invariants later in this section). If operations $op1$ and $op2$ in OZ-ASM are both applicable then one of them is chosen non-deterministically. This choice is simulated in the corresponding operation-ASM by a toggle variable *which_op*. If only one of the operations is applicable, the ASM chooses that one. If none of the operations is applicable the ASM does nothing, i.e. skips.

$$op_1 \;[]\; op_2 \qquad\qquad \Downarrow \tag{14}$$

```
if $guard(\langle\!\langle op_1 \rangle\!\rangle)$ and $guard(\langle\!\langle op_2 \rangle\!\rangle)$ and $\langle\!\langle state\_invariants \rangle\!\rangle$
then
 choose $which\_op$ in $\{first\_op, scd\_op\}$
     if $which\_op = first\_op$
     then $upds(\langle\!\langle op_1 \rangle\!\rangle)$
     else if $which\_op = scd\_op$
           then $upds(\langle\!\langle op_2 \rangle\!\rangle)$
           endif
     endif
 endchoose
else if $guard(\langle\!\langle op_1 \rangle\!\rangle)$ and $\langle\!\langle state\_invariants \rangle\!\rangle$
     then $upds(\langle\!\langle op_1 \rangle\!\rangle)$
     else if $guard(\langle\!\langle op_2 \rangle\!\rangle)$ and $\langle\!\langle state\_invariants \rangle\!\rangle$
           then $upds(\langle\!\langle op_2 \rangle\!\rangle)$
           endif
     endif
endif
```

Operation Promotion A class may contain an operation that models the application of an operation of an object instance of another class. For example, $a.op$ invokes the operation op on the object a. Assume that op is already mapped into an operation-ASM, then the invocation $a.op$ effects that the first parameter in all dynamic/external functions that are declared in op are instantiated with object a. These functions are local to the object a.

$$a.op \Rightarrow \langle\!\langle op \rangle\!\rangle(a) \tag{15}$$

Scope Enrichment Scope enrichment, $op_1 \bullet op_2$, introduces an additional level of scope for the operation op_2. As the modular structure of OZ-ASM is flattened in our mapping, a new ASM-SL function is introduced for each attribute. These functions are globally accessible within each operation-ASM. Therefore, scope enrichment can be reduced to the conjunction of both operations.

$$op1 \bullet op2 \Rightarrow \langle\!\langle op1 \;\wedge\; op2 \rangle\!\rangle \tag{16}$$

Parallel Operator The parallel composition operator is similar to the conjunction operation, but additionally models communication between both operations. $op_1 \parallel op_2$ models an operation that executes op_1 and op_2 in parallel and matches similarly named (i.e. matching apart from the '?' or '!' decoration) input and output variables of both classes. Additionally, all input/output name-matched variables of the composed operation are hidden. Hiding of a variable means that it is not visible to the environment.

The mapping of this operator is defined by its semantical definition as introduced in [19]. It is given in terms of the conjunction operator that is defined earlier (see mapping 13) plus an additional renaming of the matching input and output variables and hiding of the latter. Hiding of variables in Object-Z can be simulated in ASM by means of fresh variables that do not occur elsewhere in the specification.

Assume, we have functions $in(op)$ and $out(op)$ that provide, with the '?' or '!' decoration removed, all name-matching input and output variables of an operation. We introduce new variables, $z_1, \dots z_{n+m}$, that do not appear as free variables elsewhere in the OZ-ASM model. These are used as hidden output variables. Since we do not distinguish in ASM between output variables and internal variables, the z_i are declared, as is usual, as dynamic functions. We get

$$\begin{array}{c} op_1 \;\parallel\; op_2 \\ \Downarrow \\ \langle\!\langle \; \big(op_1[x_1!/x_1?, \dots, x_n!/x_n?] \;\wedge\; op_2[y_1!/y_1?, \dots, y_m!/y_m?]\big) \\ \qquad [z_1/x_1!, \dots, z_n/x_n!, z_{n+1}/y_1!, \dots, z_{n+m}/y_m!] \;\rangle\!\rangle \end{array} \tag{17}$$

$$\text{where } \begin{array}{l} in(op1) \cap out(op2) = \{x_1, \dots, x_n\} \\ in(op2) \cap out(op1) = \{y_1, \dots, y_m\} \end{array}$$

State Invariants OZ-ASM state invariants enable system behaviour to be modelled abstractly. Invariants allow certain states to be avoided without having to explicitly include the required constraints in the specification of the operations.

Basically, invariants involve attributes and are predicates that need to be satisfied by the pre- and post-state of any operation. One of their effects is to help determine if an operation is applicable or not.

If we can show that the specified initial state satisfies the invariant then it is sufficient to show that for each operation the invariant is not violated in the post-state. If this is guaranteed, we can be sure that all pre-states (i.e. all reachable states) satisfy the invariant too.

To guarantee that the post-state of an operation satisfies the invariant, we add an extra guard to each operation-ASM. In both Object-Z, and OZ-ASM, this extra condition is implicitly expressed in terms of pre-values (unprimed) and post-values (primed) of attributes; however, within the operation-ASM any guard that models the extra condition may not depend on post-values. Therefore, we simply substitute each primed attribute with the update-value that will be assigned when the transition is fired. This is illustrated in the following example:

```
A
  ┌──────────────          ┌─ increase ─────
  │ n : ℕ                  │ Δ(n)
  ├────                    ├────
  │ n < 10                 │ n := n + 1
  └──────────────          └──────────────
```

The invariant on the post-state of every operation is given as $(n' < 10)$. Each primed variable has to be substituted by its post-state evaluation, i.e. $(n' < 10)[n+1/n'] = (n+1 < 10)$. Therefore, we get

```
transition increase ==
   if (n + 1 < 10)
   then n := n + 1
   endif
```

In general, we get an additional guard for each operation in the class (in the example above the extra guard is $(n+1 < 10)$). In the next section we illustrate the application of the mapping rules introduced above.

4 The Operation Transformation, an Example

In this section, we show how the given mapping rules can be applied to transform OZ-ASM operations (which are defined via operation operators) into an operation-ASM. As an example we use the operation *insertKey* of class *KeySystem* that is introduced in Section 2. We assume that an algorithm for substitution is given and that the variable *self* within a class is equal to the object it refers to, i.e. $\langle\langle obj.self \rangle\rangle = self(obj) = obj$.

To begin, we show how to map the state invariant of the class *KeySystem*, namely $\forall k : keys \bullet k.rooms \subseteq rooms$. This invariant can be transformed into the following ASM-SL expression:

```
forall k in Key :
      Key_rooms(k, r) = true implies KeySystem_rooms(r) = true
```

As discussed in Section 3.3, it is sufficient to show that the invariant is satisfied in the initial state and for all post-states of the operations. The operation *insertKey* comprises the operations *supplyId* and *unlock* of class *Room* and operations *accessGranted* and *accessDenied* of class *Key*. Of these only the operation *unlock* has a non-empty update, namely $lock := false$. Since this update does not affect the state invariant given above, we do not have to add an additional guard to the operation-ASM *insertKey* to ensure the satisfiability of the invariant.

For the sake of readability, in the following we omit the brackets $\langle\!\langle . \rangle\!\rangle$ within the intermediate steps of the transformation.

$$\begin{array}{l}
insertKey \mathrel{\widehat=} [r? : rooms;\ k? : keys] \ \bullet \\
\qquad r?.supplyId \\
\qquad \parallel\ (k?.accessGranted\ \wedge\ r?.unlock \\
\qquad\qquad [\!]\ k?.accessDenied)
\end{array}$$

$$\Downarrow \qquad \text{via (17)}$$

$$\begin{array}{l}
insertKey \mathrel{\widehat=} [r? : rooms;\ k? : keys] \ \bullet \\
\qquad r?.supplyId \\
\qquad \wedge\ \big(k?.accessGranted\ \wedge\ r?.unlock \\
\qquad\qquad [\!]\ k?.accessDenied\big)[rm!/rm?]
\end{array}$$

$$\Downarrow \qquad \text{via (15), (12)}$$

$$\begin{array}{l}
insertKey \mathrel{\widehat=} [r? : rooms;\ k? : keys] \ \bullet \\
\qquad \texttt{if}\ \ rm! = self(r?)\ \texttt{then skip} \\
\qquad \wedge\ \big(k?.accessGranted\ \wedge\ r?.unlock \\
\qquad\qquad [\!]\ k?.accessDenied\big)[rm!/rm?]
\end{array}$$

$$\Downarrow \qquad \text{via } (self(x) = x)$$

$$\begin{array}{l}
insertKey \mathrel{\widehat=} [r? : rooms;\ k? : keys] \ \bullet \\
\qquad \texttt{if}\ \ rm! = r?\ \texttt{then skip} \\
\qquad \wedge\ \big(k?.accessGranted[rm!/rm?]\ \wedge\ r?.unlock[rm!/rm?] \\
\qquad\qquad [\!]\ k?.accessDenied[rm!/rm?]\big)
\end{array}$$

$$\Downarrow \qquad \text{via (15), (12), (substitution)}$$

$$\begin{array}{l}
insertKey \mathrel{\widehat=} [r? : rooms;\ k? : keys] \ \bullet \\
\qquad \texttt{if}\ \ rm! = r?\ \texttt{then skip} \\
\qquad \wedge\ \big((\texttt{if}\ \ key\_rooms(k?, rm!) = true\ \texttt{then skip} \\
\qquad\quad \wedge\ \texttt{if}\ \ room\_locked(rm!) = true\ \texttt{then}\ \ room\_locked := false) \\
\qquad\quad [\!]\ (\texttt{if}\ \ key\_rooms(k?, rm!) = false\ \texttt{then skip}\ )\big)
\end{array}$$

$$\Downarrow \qquad \text{via (13)}$$

$$\begin{array}{l}
insertKey \mathrel{\widehat=} [r? : rooms;\ k? : keys] \ \bullet \\
\qquad (\texttt{if}\ \ rm! = r?\ \texttt{and} \\
\qquad\qquad key\_rooms(k?, rm!) = true\ \texttt{and}\ \ room\_locked(rm!) = true \\
\qquad\ \ \texttt{then}\ \ room\_locked := false) \\
\qquad [\!] \\
\qquad (\texttt{if}\ \ rm! = r?\ \texttt{and} \\
\qquad\qquad key\_rooms(k?, rm!) = false \\
\qquad\ \ \texttt{then skip}\ )
\end{array}$$

$\Downarrow$ via (3), (12), (16), (14)

```
external function $k\_in : KeyType$
   with $k\_in$ in $Key$
external function $r\_in : RoomType$
   with $r\_in$ in $Room$

transition $insertKey$ ==
     if $rm\_out = r\_in$ and
     $key\_rooms(k\_in, rm\_out) = true$ and $room\_locked(r\_in) = true$
        and $key\_rooms(k\_in, rm\_out) = false$
     then
      choose $which\_op$ in $\{first\_op, scd\_op\}$
         if $which\_op = first\_op$
         then $room\_locked(r\_in) := false$
         else if $which\_op = scd\_op$
              then skip
              endif
         endif
      endchoose
     else if $rm\_out = r\_in$ and
              $key\_rooms(k\_in, rm\_out) = true$ and $room\_locked(r\_in) = true$
         then $room\_locked(r\_in) := false$
         else if $rm\_out = r\_in$ and
                 $key\_rooms(k\_in, rm\_out) = false$
              then skip
              endif
         endif
     endif
```

5 Conclusion and Future Work

We have shown in this paper how language integration can be used to define a subset of a specification language in order to tailor it for automated tool support. We introduced the integrated language OZ-ASM, which combines Object-Z with ASM transition rules. OZ-ASM enables the user to model systems in a state transition based fashion, a style that can also be adopted by Object-Z users if operation predicates are modelled in a canonical form. The syntax of ASM transition rules provides a clear definition of this canonical form, which ensures that primed attributes (i.e. variables in the next state) depend only on non-primed attributes (i.e. variables in the current state). State transition systems can interface with various analysis tools, such as state transition based model checkers like SMV.

We showed by means of a set of mapping rules how OZ-ASM can be transformed into a set of ASMs which in turn can be automatically compiled into

SMV code. The interface from ASM and the ASM Workbench tool environment to the SMV model checker is already available. Therefore, the results of this paper provide the theoretical basis for an interface from the integrated language OZ-ASM to the model checker SMV. Clearly, the interface needs to be implemented in order to provide automated tool support; this is future work.

Our approach complements the work of others in which either a process-algebra based model checker or SAT solvers are interfaced with a Z-based language. Each of these approaches have their special merits for particular applications.

Future work involves the completion of the list of mapping rules for operation operators (e.g. the distributed operation operators) and OZ-ASM expressions and data structures (e.g. sequences). Based on the mapping rules, a transformation algorithm needs to be implemented. We will also investigate the use of other analysis tools that deal with state transition systems. Especially, other model checkers could be interfaced using ASM as an intermediate language (e.g.NuSMV [18], VIS [22], MDG-Tool [17])[4]. Abstraction and decomposition techniques have to be developed which target the limitations of model checking with respect to the model size. Furthermore, we will attempt to develop (informal or formal) rules for mapping Object-Z into OZ-ASM so that model checking based on transition systems can be directly available for Object-Z.

Acknowledgements Thanks to Graeme Smith for his initial inspiration of this work and his useful comments on earlier drafts of this paper.

References

[1] G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proc. of 6th Int. Conference for Tools and Algorithms for the Construction and Analysis of Systems, (TACAS 2000)*, vol. 1785 of LNCS, Springer-Verlag, 2000.

[2] G. Del Castillo. The ASM Workbench. PhD thesis, Department of Mathematics and Computer Science of Paderborn University, Germany, 2000.

[3] R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z.* Macmillan Press, 2000.

[4] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier Science Publishers, 1990.

[5] C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway and K. Taguchi editors, *Proceedings of the 1st International Conference on Integrated Formal Methods (IFM'99)*, pages 315–334. Springer-Verlag, 1999.

[6] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, Oct 1997.

[4] As shown in [23], the transformation from ASM-SL into the SMV language can be easily adapted to interface other tools that are based on transition systems.

[7] W. Grieskamp. A computation model for Z based on concurrent constraint resolution. In *ZB2000 – International Conference of Z and B Users*, September, 2000.

[8] Y. Gurevich. May 1997 Draft of the ASM Guide. Technical report, University of Michigan EECS Department, 1997.

[9] Y. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 2000.

[10] G. Holzmann. Design and validation of protocols: A tutorial. In *Computer Networks and ISDN Systems*, volume XXV, pages 981–1017, 1993.

[11] G. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[12] D. Jackson. Nitpick: A checkable specification language. In *Proc. of the First ACM SIGSOFT Workshop on Formal Methods in Software Practice*, pages 60–69, 1996.

[13] D. Jackson, I. Schechter and I. Shlyakhter. Alcoa: the Alloy constraint analyser. In *Int. Conf. on Software Engineering*, 2000.

[14] J. Jacky and M. Patrick. Modelling, checking and implementing a control program for a radiation therapy machine. In R. Cleaveland, D. Jackson, editors, *Proc. of the First ACM SIGPLAN Workshop on Automated Analysis of Software(AAS'97)*, pages 25–32, 1997.

[15] G. Kassel and G. Smith. Model checking Object-Z classes: Some experiments with FDR. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, IEEE Computer Society Press, 2001 (to appear).

[16] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[17] F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny. Multiway Decision Graphs for automated hardware verification. In *Formal Methods in System Design*, 10(1), 1997.

[18] A. Cimatti, E.M. Clarke, F. Giunchiglia and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *11th Conference on Computer-Aided Verification (CAV'99)*, vol. 1633 of LNCS, Springer-Verlag, 1999.

[19] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, 2000.

[20] J.M. Spivey. *The Z Notation - A Reference Manual.* Prentice Hall, 1992.

[21] S. Valentine. The programming language Z--. *Information and Software Technology*, volume 37, number 5-6, pages 293–301, May-June, 1995.

[22] The VIS Group. VIS: A System for Verification and Synthesis. In R. Alur and T. Henzinger, editors, *8th Int. Conf. on Computer Aided Verifaction, (CAV'96).* vol. 1102 of LNCS, Springer-Verlag, 1996.

[23] K. Winter. *Model Checking Abstract State Machines.* PhD thesis, Technical University of Berlin, Germany, http://edocs.tu-berlin.de/diss/2001/winter_kirsten.htm, 2001.

[24] K. Winter. Model checking with abstract types. In S. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[25] P. Zave. Formal description of telecommunication services in Promela and Z. In *Calculational System Design, Proc. of the Nineteenth International NATO Summer School.* IOS Press, 1999.

A Model checking OZ-ASM: an example

The following example shows that model checking can provide useful support even for small systems.

A

$upperBound : \mathbb{N}$

$upperBound = 10$

$lower, upper : 1..upperBound$

$lower \neq upper$

$\textsc{Init}$

$lower = 1$
$upper = 10$

$move$

$\Delta(lower, upper)$

```
if  $lower < upper$
then $lower := lower + 1$
     $upper := upper - 1$
```

Suppose that we suspect that the predicate $lower < upper$ is a system invariant. An informal argument by structural induction in support of this hypothesis is as follows:

Initial step: as declared in the $\textsc{Init}$ schema, $lower < upper$ is true initially.

Induction step: $lower < upper$ is declared to be a precondition of the only operation, *move*. Furthermore, $lower \neq upper$ is declared in the state schema as a given invariant. Hence if the *move* operation is applicable and occurs, as the values of both *lower* and *upper* change by exactly 1 (the value of *lower* is increased by 1, the value of *upper* is decreased by 1), and as the invariant $lower \neq upper$ must be true after the operation (else the operation would not have been enabled and could not have occurred), after the operation $lower < upper$ must still be true. Hence by structural induction the predicate $lower < upper$ is a system invariant.

The flaws in this reasoning might be revealed by careful thought and reflection; however, tool support by means of a model checker is an excellent way to confirm or refute such hypotheses.

Applying the transformation steps introduced in Section 3, we generate the following ASM-SL model. (Because there is only one class in our example specification, we don't bother to add the class name to the local names.)

```
static function $upperBound == 10$

dynamic function $lower : INT$
      with $lower \in \{1..upperBound\}$
      initially 1

dynamic function $upper : INT$
      with $upper \in \{1..upperBound\}$
      initially 10

transition $move ==$
      if $(lower < upper)$
            and $(lower \neq upper)$
            and $(lower + 1 \neq upper - 1)$
      then $lower := lower + 1$
           $upper := upper - 1$
      endif
```

This ASM-SL model can be automatically transformed into SMV code. Using this code we can now check if *lower* $<$ *upper* is a system invariant, or as specified in CTL (see [4]), (`AG` *lower* $<$ *upper*). The result is a counter-example given as a sequence of states of the system that lead to a violation of the CTL formula.

```
-- specification AG s._lower < s._upper is false
-- as demonstrated by the following execution sequence
state 1.1: s._lower = 1
           s._upper = 10
state 1.2: s._lower = 2
           s._upper = 9
state 1.3: s._lower = 3
           s._upper = 8
state 1.4: s._lower = 4
           s._upper = 7
state 1.5: s._lower = 5
           s._upper = 6
state 1.6: s._lower = 6
           s._upper = 5

resources used:
user time: 0.04 s, system time: 0.02 s
BDD nodes allocated: 2241
Bytes allocated: 1245184
BDD nodes representing transition relation: 161 + 10
```